Una Herramienta para Optimizar el Comportamiento de los Agentes Inteligentes en Videojuegos mediante Computación Evolutiva

A Tool to Optimize the Behavior of Intelligent Agents in Video Games through Evolutionary Computing

Trabajo Fin de Grado
Curso 2020–2021

Autores
Alejandro Ansón Alcolea
Adrián Ogáyar Sánchez
Maikel Jesús Spranger Hierro

Directores
Federico Peinado Gil
Maximiliano Miranda Esteban

Grado en Desarrollo de Videojuegos
Grado en Ingeniería de Sistemas
Facultad de Informática
Universidad Complutense de Madrid

Una Herramienta para Optimizar el Comportamiento de los Agentes Inteligentes en Videojuegos mediante Computación Evolutiva

A Tool to Optimize the Behavior of Intelligent Agents in Video Games through Evolutionary Computing

Trabajo Fin de Grado

Autores

Alejandro Ansón Alcolea
Adrián Ogáyar Sánchez
Maikel Jesús Spranger Hierro

Directores

Federico Peinado Gil
Maximiliano Miranda Esteban

Convocatoria: *Junio 2021*

Grado en Desarrollo de Videojuegos
Grado en Ingeniería de Sistemas
Facultad de Informática
Universidad Complutense de Madrid

15 de junio de 2021

Agradecimientos

Alejandro Ansón Alcolea

En primer lugar me gustaría agradecer a Fede por aceptar en un principio dirigir mi TFG a pesar de no tener una buena idea inicial sobre qué hacer, y por permitir aumentar las plazas en dos para incluir a mis compañeros Maikel y Adri. A ellos también les tengo que agradecer no solo el buen trabajo hecho sino también el buen ambiente que hemos tenido durante estos meses. Echaba bastante de menos esto desde que entré a la universidad, que fue cuando nos conocimos.

También tiene un lugar especial en este agradecimiento mi familia, que han estado conmigo e hicieron todo lo que podían para ayudarme; permitiendo que pueda trabajar incluso cuando estábamos de viaje, prestándome ese apoyo incondicional desde hace años y sin el cual nunca habría llegado hasta donde estoy.

No me puedo olvidar tampoco de mis mejores amigos fuera de la universidad, que me han aguantado todo el año con estrés y llamadas constantes; pasando tiempo conmigo mientras trabajaba haciendo más ameno todo este viaje.

Durante mi periodo de estudio en la universidad he conocido a gente maravillosa tanto en Ingeniería Informática, como en Desarrollo de Videojuegos.

Entre ellos no sólo se encuentran mis compañeros actuales de TFG, si no también mi primer grupo de amigos universitarios, que a pesar de poder tratarnos como burros entre nosotros, siempre estará ahí; también mi grupo de desarrollo de videojuegos SpicySeed, que desde primer y segundo año de carrera hemos sido inseparables, haciendo mucho más fácil sobrellevar estos años.

Por último, me gustaría agradecer a todos los profesores que he conocido a lo largo de la carrera, por enseñarme y prepararme para la vida laboral.

Muchas gracias a todos por hacer que estos años de vida universitaria hayan sido realmente buenos.

Adrián Ogáyar Sánchez

Cualquiera que me conozca sabe que no soy la persona más expresiva del mundo. Posiblemente tampoco la más sentimental. Sin embargo no puedo dejar este espacio en blanco. No sería justo. Hoy no estaría aquí escribiendo estas palabras, terminando la memoria del TFG, si no fuera por esa gente que me apoyó cuando más lo necesitaba.

Me gustaría empezar agradeciendo a mi familia, que siempre hizo todo lo posible por ayudarme. Y en especial a mi abuelo, sin el cual no sería la persona que soy hoy día. Sé que se alegraría muchísimo de ver que uno de sus nietos ha terminado una carrera universitaria. Gracias por enseñarme tanto. También me gustaría agradecer a mi madre por darme una segunda oportunidad que jamás merecí, también por todos esos sacrificios y esfuerzos para que nunca me faltara nada. Espero que todo esto sirva para que le pueda comprar esa casa en la playa que tanto ansía.

Tampoco puedo olvidarme de Ainhoa, cuya dedicación y esfuerzo me motivaron a dar el máximo de mí mismo y me hizo darme cuenta de que de verdad era capaz de estudiar y sacar buenas notas. Gracias también a Pepe, mi profesor de Física de bachillerato. que me hizo esforzarme más allá de ese máximo y me hizo plantearme que, quizás, podría estudiar una carrera.

Por otro lado me es inevitable pensar en toda esa gente que, aunque no me ha hecho llegar hasta aquí, sí que hizo que el camino fuera muchísimo más divertido. Entre toda esta gente, es obligatorio mencionar a mis "panas", esa panda de locos que conocí el primer año de carrera, dos de los cuales están aquí conmigo escribiendo esta memoria. Gracias por todos esos días y noches de partidas eternas, incluso durante exámenes. Gracias por todas esas horas muertas en la biblioteca jugando a la *Switch*. Gracias por esas felicitaciones de cumpleaños, Navidad y Año Nuevo a destiempo. Gracias por estar ahí. Espero veros pronto para celebrarlo.

Por supuesto también debo hablar de la gente maravillosa que he conocido en mi carrera. Lamento que la pandemia nos hiciera distanciarnos. De todos ellos, creo que sin duda debo agradecer especial y personalmente a Vero. Gracias por hacerme despertar, por hacerme ver que debía seguir esforzándome si quería alcanzar mis metas. Gracias por estar ahí durante este largo y horrible año de pandemia. Y por ser tan buena profesora de inglés, sin ti no habría encontrado mi trabajo actual. Gracias por todas esas noches eternas hablando de la vida. Espero que todas esas reflexiones y divagaciones te guíen tal y como han hecho conmigo. Lucha por tus sueños. Espero que dentro de muchos años volvamos a coincidir, pues eso, probablemente, significará que ambos logramos alcanzarlos. *You're the best.*

Por último, pero no menos importante, quiero agradecer a Fede su trabajo tutorizándonos, dándonos la libertad para hacer el TFG sin restricciones. Sin él, todo esto nunca hubiera sido posible.

Maikel Jesús Spranger Hierro

Aquella primera vez que tuve un ordenador entre manos hace alrededor de 20 años y pude jugar a mi primer videojuego me quede fascinado por lo que veía, quise saber como funcionaba y poder crear mis propios videojuegos. A día de hoy, después de muchos años de estudio y muchas pruebas puedo decir que sé cómo funcionan en general y me veo más cerca de poder cumplir este sueño que he tenido por un largo tiempo.

No puedo agradecer lo suficiente a mi familia por apoyarme todo este tiempo, a mi madre que siempre estuvo apoyándome, llevándome a hacer cursos y empujándome a continuar con la Informática después de algunos resbalones por el camino; a mi padre que, aunque quería que siguiera sus pasos con la Mecánica, nunca me puso pegas en lo que yo había elegido para mi futuro y siempre me facilitó todo lo que pude necesitar para avanzar en este camino y a mi querida "hermanita" mayor, con la que siempre puedo contar para cualquier cosa y a la que le puedo contar siempre cuando suspendo un examen... y también cuando lo apruebo, claro.

Le doy las gracias a Fede por dirigirnos y ayudarnos a mantener el trabajo continuo durante todo el plazo, así como todas las sugerencias y comentarios que nos permitieron reflexionar sobre nuestro proyecto.

Quiero agradecer a mis amigos de la uni (ese mismo grupo de locos, máquinas, fieras, mastodontes... que mencionamos todos por aquí) que nos conocimos en el primer año. Espero que sigamos conectados por mucho tiempo, sobretodo para esas felicitaciones de cumpleaños y año nuevo que ellos entenderán, para muchas más quedadas y mucha... ¿cultura?.

Como estudie en bachillerato nocturno con Adri me veo obligado a decir "grande Pepe", un profesor que nunca olvidaré y que me demostró que era más inteligente de lo que pensaba y que podía seguir adelante. Además de agradecer a Fernando, mi profesor de lengua y jefe de estudios, que -ya habiendo empezado el curso al que sólo fui por curiosidad de mi madre sobre el bachillerato nocturno- no tardó más de 10 minutos en convencerme de que podía continuar mis estudios y terminarlos, cosa en la que puedo confirmar que no se equivocaba.

No puedo dejar fuera de este agradecimiento a esos amigos "online" de diferentes países con los que he compartido mucho, tanto en videojuegos, como en chat de voz a las tantas de la madrugada, hablando de políticas exteriores, ética, progresión en videojuegos y de cualquier cosa que se nos ocurriera a esas horas. En especial tengo que agradecer a Sid -o como lo conocí por primera vez *Itsurugi*- quién me ayudo a clavar la entrevista de código para mi primer trabajo como programador con muchos consejos sobre *Kotlin*.

Y a todo el team [R][P][B]ocket: *Love you guys, be back really soon! Keep up the good work.*

Resumen

El equilibrado en el diseño de videojuegos es un problema complejo existente desde que aparecieron los primeros títulos. Un videojuego bien planteado, con mecánicas y dinámicas interesantes, pero mal equilibrado, donde, por ejemplo, la dificultad no es proporcional a la destreza del jugador, o donde los enemigos no saben aprovechar sus oportunidades para combatir al protagonista, acabará fracasando como experiencia de entretenimiento.

Por eso este proyecto propone una herramienta automática, EvoUnity, que permite optimizar el comportamiento de los agentes inteligentes en videojuegos, principalmente orientada a equipos pequeños de desarrolladores independientes que no disponen de mucho presupuesto o tiempo para equilibrar o ajustar la inteligencia artificial de sus juegos con metodologías clásicas más costosas.

Esta herramienta extiende el entorno de desarrollo de videojuegos *Unity*, muy popular entre este tipo de desarrolladores. Hoy día, alrededor del 50 % de los videojuegos para PC, móvil y consolas son desarrollados con esta tecnología. *Unity* es muy popular gracias a su versatilidad al permitir desplegar juegos en múltiples plataformas, a su integración con servicios de anuncios, compras y analíticas, y a su facilidad de uso, pues es una herramienta intuitiva con abundante documentación, lo que también facilita su aprendizaje.

EvoUnity busca optimizar el comportamiento de los agentes inteligentes de cualquier videojuego, independientemente de su genero o características, en base a unos resultados esperados de rendimiento medibles y bien definidos en cada partida, utilizando técnicas de computación evolutiva en el proceso de optimización.

Para probar esta herramienta se utilizan dos proyectos de código abierto, concretamente el juego de defensa de torres *Warrior Defense* y el prototipo de juego de estrategia en tiempo real *MicroRTS*. El primero sirve como prueba de integración de la herramienta en un juego real, y el segundo ayuda a generalizar del software, al presentar mayor complejidad y variedad de elementos a tener en cuenta para el ajuste. En ambos se han conseguido resultados prometedores que permiten proponer la herramienta como un recurso útil ante la comunidad de desarrolladores.

Palabras clave

Algoritmos Evolutivos, Inteligencia Artificial, Herramientas para Videojuegos, *Unity*, Jugadores Automáticos.

Abstract

Balance in videogames has been an issue since the first titles appeared. A well-designed videogame, with interesting mechanics and dynamics, but poorly balanced, where, for example, the difficulty is not proportional to the player's skill, or where the enemies do not know how to take advantage of their opportunities to fight the protagonist, will end up failing as an entertainment experience.

That is why this project proposes an automatic tool, EvoUnity, which allows to optimize the behavior of intelligent agents in video games, mainly oriented to small teams of independent developers who do not have much budget or time to balance or adjust the artificial intelligence of their games with more expensive classical methods.

This tool extends the video game development environment *Unity*, which is very popular among this type of developers. Today, about 50 % of video games for PC, mobile and consoles are developed with this technology.

Unity is very popular thanks to its versatility in allowing games to be deployed on multiple platforms, its integration with ad, shopping and analytics services, and how easy it is to use, as it is an intuitive tool with abundant documentation, which also makes it easy to learn.

EvoUnity seeks to optimize the behavior of intelligent agents in any video game, regardless of its genre or characteristics, based on measurable and well-defined expected performance results in each game, using evolutionary computation techniques in the optimization process.

Two open source projects are used to test this tool, concretely the tower defense game *Warrior Defense* and the real-time strategy game prototype *MicroRTS*. The first one serves as a test for the integration of the tool in a real game, and the second one helps to generalize the software, as it presents a greater complexity and variety of elements to be taken into account for the adjustment. In both cases, promising results have been achieved, allowing the tool to be proposed as a useful resource to the developer community.

Keywords

Evolutionary Algorithms, Artificial Intelligence, Video Game Tools, *Unity*, Automatic Players.

Índice

Índice de figuras

Capítulo 1

Introducción

En este capitulo se introduce el tema del TFG, el alcance y el propósito de realizarlo. Además se presenta la estructura de la memoria.

El equilibrio de los componentes de un videojuego lleva siendo un problema para los diseñadores desde hace décadas, prácticamente desde la aparición de los primeros videojuegos con cierta complejidad. Que un arma sea demasiado poderosa, que un enemigo tenga demasiada vida, el alcance de los ataques de una unidad, etc. Todos estos parámetros pueden hacer que un videojuego resulte demasiado fácil o difícil para el jugador, resultando en una experiencia insatisfactoria.

Por ello, es común que los desarrolladores de videojuegos inviertan cientos de horas, ya sea mediante un equipo interno de *QA (Quality assurance)*, contratando *testers* u organizando periodos de pruebas, públicas o privadas, del software en producción, conocidas como alfas o betas, para asegurarse no sólo de que el juego funciona, sino que también está equilibrado. En muchos casos, como en la mayoría de los juegos para un jugador, un arma demasiado fuerte o débil, o un enemigo con demasiada vida puede no suponer un problema si el jugador tiene la opción de elegir otro arma o si ese enemigo, aunque difícil de matar, no supone un reto demasiado grande como para frustrar al jugador. Sin embargo, en el modo de juego jugador contra jugador (JcJ), es imprescindible que exista un equilibrio, dado que si un arma o unidad es demasiado poderosa los jugadores estarán "obligados" a usarla o se encontrarán en desventaja frente al resto, lo que limitará las opciones de los jugadores y, probablemente, dará lugar a partidas repetitivas donde se reproducen las mismas estrategias.

Por tanto, el *testing* del equilibrio de los videojuegos es un proceso que supone una inversión de recursos importante pero necesaria, sin el cual la mayoría de los videojuegos estarían abocados al fracaso.

Además, cuanto más complejo es un juego, mayor es la dificultad para probar todas las posibilidades que ofrece, requiriendo mucho más tiempo y, en ocasiones, obligando a los desarrolladores a lanzar una beta para obtener

el mayor número de jugadores y asegurarse de que se prueban todas las combinaciones que ofrece el juego.

Este problema crece en gran medida cuando se tiene que equilibrar la inteligencia artificial de un juego. Este aspecto es uno de los más críticos respecto al balance de un videojuego, y una IA mal desarrollada puede provocar altos niveles de frustración en el jugador, o incluso generar una caída en la dificultad y perder completamente la sensación de desafío. Suele ser común además el implementar múltiples niveles de dificultad, y dependiendo del tipo de juego es necesaria la modificación de la inteligencia artificial, provocando cambios inesperados en la experiencia del juego. El impacto de la IA en el balance de un videojuego va mucho más allá de parámetros básicos como el daño que puede hacer un arma (y cómo los jugadores pueden aprovecharse de ello), sino que puede suponer un cambio fundamental en los enfrentamientos y cualquier cambio sutil a los parámetros de esta pueden provocar que algo sea imposible de superar.

Esto, si bien puede no suponer un problema de recursos para estudios grandes, sí lo es para estudios pequeños que disponen de un menor presupuesto. Invertir decenas, o cientos, de miles de euros en testear un videojuego puede resultar imposible para muchos desarrolladores y, aunque los propios desarrolladores pueden ocuparse de este trabajo, esto implica gastar cientos de horas jugando y probando las distintas estrategias.

Con los motores de videojuego gratuitos (o con sistemas de comisiones) y el auge de la industria *indie*, el número de juegos de bajo presupuesto ha crecido enormemente y con ello, la necesidad de buscar nuevos enfoques para probar videojuegos sin depender de personas que los prueben, reduciendo los costes y tiempo de desarrollo.

Por ello, el objetivo de este trabajo es investigar el desarrollo de una herramienta diseñada para el motor de videojuegos *Unity*[1] que, haciendo uso de la computación evolutiva, sea capaz de probar el juego y automatizar la elección de una inteligencia artificial adecuada para la dificultad deseada por el diseñador.

1.1. Alcance

Aunque un posible enfoque sería diseñar específicamente una herramienta de *Unity* para un tipo de videojuego, el objetivo final sería crear un software capaz de funcionar con cualquier tipo de juego, independientemente del género o características de este, sin que los desarrolladores requieran grandes conocimientos de programación o de algoritmia. Sin embargo, esto implica generar una IA (inteligencia artificial) capaz de aprender a jugar a cualquier videojuego aunque sea a un nivel muy básico, algo que, si bien es un ob-

[1] `https://unity.com/`

jetivo alcanzable a largo plazo, supone una tarea muy grande y lejos de lo que este trabajo plantea, más enfocado en el testeo y en el uso de algoritmo evolutivos.

Por tanto, la herramienta se diseñará usando un juego de código abierto, *Warrior Defense*, para analizar el rendimiento de esta. Adicionalmente, se intentarán utilizar otros géneros en los que ejecutar la herramienta para comprobar su eficiencia. Por ello, es necesaria la creación de una IA capaz de jugar al videojuego seleccionado, así como el diseño de un algoritmo evolutivo que permita aleatorizar las variables y quedarse con las mejores combinaciones para obtener el resultado deseado por el diseñador.

1.2. Propósito

El problema principal que buscamos resolver es la gran carga de tiempo y recursos, tanto humanos como computacionales, que ocasiona el equilibrado de inteligencias artificiales en videojuegos. Este problema se ve acentuado en desarrolladores con poca experiencia y nuestro objetivo es ayudarles a agilizar el proceso y mejorar sus resultados, todo esto de forma automática y con poca configuración manual por parte de ellos.

1.3. Estructura del trabajo

El presente trabajo se estructura como sigue. En este Capítulo 1 se presenta el alcance y el propósito que tiene este Trabajo Fin de Grado. En el Capítulo 2 se revisa el estado de la cuestión en materia de Inteligencia Artificial usada en videojuegos o como técnica para ayudar en el desarrollo de estos. En el Capítulo 3 se revisan los objetivos en detalles, teniendo en cuenta las especificaciones correspondientes de la herramienta, teniendo en cuenta restricciones,funciones y características del usuario. En el Capítulo 4 se presenta las herramientas utilizadas al igual que la planificación que se ha seguido en el desarrollo del TFG. En el Capítulo 5 se realiza un análisis sobre los objetivos planteados y las especificaciones, también se presenta el diseño de la herramienta EvoUnity, como las cuestiones de implementación y pruebas encontradas a lo largo del desarrollo. En el Capítulo 6 se presentan todos los resultados de las pruebas realizadas en ambos juegos, al igual que la discusión sobre ellos. Por ultimo se termina con el Capítulo 7 donde se resumen las conclusiones del proyecto, y lo logrado con respecto a los objetivos.

Capítulo 2

Estado de la cuestión

En este capítulo se habla de los orígenes de la Inteligencia Artificial así como su evolución y posición en el campo de los videojuegos. También se habla de la Computación Evolutiva y los distintos algoritmos pertenecientes a este campo. Por último, se comentan las características de los dos videojuegos utilizados en este trabajo, además de *Unity*, entorno para el que se desarrolla la herramienta.

2.1. Inteligencia Artificial

La Inteligencia Artificial (IA) es la ciencia e ingeniería que hace posible la creación de máquinas inteligentes (McCarthy, 2004).

Los origines de la IA se remontan a Aristóteles, que fue el primero en plantear la idea de un conjunto de reglas que llevan a una conclusión lógica, un planteamiento muy general, pero aplicable a la IA moderna. Siglos después de esta idea, Alan Turing empezó a trabajar en el desarrollo de la primeras computadoras y con ellos ideó los conceptos de lo que sería la IA moderna, además de diferentes acercamientos teóricos al campo de la IA (aunque en aquellos tiempos este campo ni siquiera existía) como el Test de Turing (Turing, 1950) o la primera computadora capaz de jugar al ajedrez. Este programa nunca se ejecutó en una computadora debido a las limitaciones computacionales de la época, pero Turing decidió probar el algoritmo manualmente ejecutando los cálculos que la máquina realizaría para determinar los movimientos y, aunque perdió, fue una revolución por mostrar que, teóricamente, una computadora podría ser capaz de jugar al ajedrez.

Aun así, el primer trabajo en el campo de la IA fue el de Warren McCulloch y Walter Pittis en 194, con el primer modelo de neurona artificial, que consistía en un algoritmo que replicaba el funcionamiento de una neurona (McCulloch y Pitts, 1943). Este algoritmo recibe unos datos de entrada, los procesa, y envía la salida a la siguiente neurona del mismo modo que una neurona obtiene información desde las dendritas (*inputs*) y envía la infor-

mación procesada a través del axón (*output*). Este modelo se sigue usando actualmente en Redes Neuronales, uno de los algoritmos más extendidos en el campo del aprendizaje máquina.

Aunque hubo avances en el campo de la IA en los años siguientes, no fue hasta 1956 cuando el término "Inteligencia Artificial" fue acuñado en la Conferencia de Dartmouth. Esta conferencia se realizó con el objetivo de asentar los cimientos de la IA y, durante semanas, los miembros lanzaron ideas sobre lo que la IA sería en el futuro. Desafortunadamente, las ideas propuestas resultaban demasiado ambiciosas para la tecnología de la que disponían y esto causó que muchos proyectos fueran abandonados, lo que estancó las investigaciones.

Gracias a la invención de los Sistemas Expertos, el interés en la IA resurgió y las investigaciones se retomaron, esta vez con un poder computacional superior, lo que permitió hacer grandes avances, imposibles anteriormente.

En 1985, IBM inició la creación de Deep Blue, un supercomputador capaz de jugar al ajedrez, que finalmente fue probada en 1996 contra Garri Kaspárov, el antiguo Campeón del Mundo, siendo capaz de ganar la primera partida (aunque perdió finalmente 4-2). Aun así, al año siguiente IBM aplicó numerosas mejoras y Deep Blue superó a Kasparov en una revancha, convirtiéndose en la primera IA capaz de ganar a un Campeón del Mundo de Ajedrez[1].

En las últimas décadas, la IA ha sido sin ninguna duda el centro de atención de investigadores, compañías y el público general. Esto ha conducido a numerosos avances y nuevas ramas de investigación, así como al desarrollo de técnicas más avanzadas. Entre todas estas ramas, destaca el Aprendizaje Profundo, en la que se han realizado grandes progresos, lo que ha llevado la IA a prácticamente todos los dispositivos electrónicos.

Dentro de la IA, hay una rama llamada Inteligencia Computacional (IC). Esta rama investiga y desarrolla algoritmos de IA inspirados por el funcionamiento de la vida par resolver problemas complejos que no pueden ser resueltos por los modelos matemáticos tradicionales.

Los tres pilares fundamentales de la IC son las Redes Neuronales, basados en las neuronas, los Sistemas Difusos, basados en el lenguaje humano, y la Computación Evolutiva, basados en la Teoría de la Evolución.

Aún así, la IC es un campo relativamente nuevo y se encuentra en constante evolución, por ello actualmente incluye otros algoritmos y ramas de estudio como la Vida Artificial, centrada en el estudio de sistemas artificiales con un comportamiento similar a los seres vivos (Langton et al., 1991), o las Redes Neuronales Endocrinas, que proponen nuevos enfoques basados en la sinergia entre los sistemas nervioso y endocrino para crear Redes Neuronales complejas (Xu y Wang, 2011).

[1] `https://web.archive.org/web/20080701232743/http://www.research.ibm.com/deepblue/watch/html/c.shtml`

2.1.1. Técnicas de Inteligencia Artificial

Desde su origen, la IA ha intentado resolver multitud de problemas en áreas muy diversas y, debido a esto, nuevos métodos de entrenamiento y algoritmos han sido desarrollados centrados en resolver uno o más dilemas con los que técnicas más antiguas tenían dificultades.

Estas técnicas varían mucho en función de la información disponible en el contexto. *Hard Computing* es la rama que trabaja con datos exactos, generando resultados precisos con modelos deterministas. Por otro lado, *Soft Computing* trabaja con información imprecisa, incierta, a menudo incompleta o ambigua, creando modelos estocásticos. Debido a que este trabajo es sobre videojuegos, que trabajan con información imperfecta, nos centraremos en técnicas de *Soft Computing*.

Dentro del *Soft Computing* hay muchas técnicas de IA centradas en resolver diferentes problemas desde distintos ángulos. Tal y como se muestra en Chen et al. (2008), algunas de las técnicas más relevantes son:

- Redes Neuronales Artificiales (ANN): Una de las técnicas más usadas. Las Redes Neuronales consisten en la simulación del comportamiento de una neurona. Esta simulación se realiza usando vectores de nodos, que representan las neuronas, y matrices de pesos, que son las variables para "aprender" en el entrenamiento y conectan los nodos simulando la sinapsis en los cerebros biológicos. Debido a su escalabilidad, han surgido nuevas técnicas basadas en ANN para ser usadas en distintos ámbitos, como las Redes Neuronales Convolucionales (CNN), que permiten el reconocimiento de audio y vídeo.

- Razonamiento Basado en Casos (CBR): Usa una base de datos de casos previos para encontrar el caso más similar al problema actual a resolver. Tras encontrarlo, el CBR lo adapta al nuevo problema, lo evalúa y, si pasa el test, la solución es usada y añadida a la base de datos para futuras consultas.

- Computación Evolutiva (EC): Usa algoritmos inspirados en la naturaleza. Es un campo muy amplio con múltiples enfoques.

- Sistemas Difusos (FS): Usan valores entre 0 y 1 para representar la pertenencia a una clase. Esto permite crear funciones "vagas" como si fueran expresiones del lenguaje natural.

- Aprendizaje por Refuerzo (RL): Basado en la Teoría del Conductismo, esta técnica consiste en ofrecer una recompensa cuando la máquina transita de un estado a otro, buscando maximizar la recompensa.

- Árbol de búsqueda de Monte Carlo (MCTS): Usado principalmente en juegos, tanto de mesa como videojuegos, consisten en la creación de

un árbol donde los nodos representan un estado del juego y tienen un valor que indica la probabilidad de victoria a partir de ese estado. A la hora de determinar el movimiento a realizar, busca el nodo con el mejor valor dentro del árbol.

- Árbol de Decisión (DT): Consisten en un árbol que define una estructura similar a un algoritmo basado en *if-else*. Son comúnmente usados en videojuegos para la creación de IA y en Teoría de Juegos.
- Árbol de Comportamiento (BT): Haciendo uso de una estructura de árbol, son capaces de crear comportamientos complejos. Se usan principalmente para crear IA en videojuegos.

Generalmente, estas técnicas se combinan, dando lugar a sistemas más complejos, como las Redes Neuronales Evolutivas, fusionando ANN y EC, o el uso del RL en el entrenamiento de ANN.

Para este trabajo, nos centraremos en la EC para testear videojuegos y DT para crear la IA. Hablaremos de ello con más detalle en Sección 2.2 y Subsección 2.1.4 respectivamente.

2.1.2. Inteligencia Artificial para Videojuegos

La IA siempre ha estado ligada a los videojuegos desde el origen de los mismos, *Nim* fue el primer videojuego que hizo uso de una IA simple, mediante teoría matemática, que permitía determinar los movimientos necesarios para alcanzar la victoria a partir del estado actual del juego, llegando a ganar incluso a oponentes humanos expertos.

A partir de finales de los años 50, la investigación de IA capaces de jugar a juegos de mesa comenzó. La IA de Arthur Samuel, desarrollada en 1959, era capaz de jugar a las damas contra jugadores amateur. Esto llevó a la creación de Chinook en 1989, que en 1994 logró el título de Campeón de Damas Humano-Máquina, aunque nunca ganó ninguna de las partidas que jugó contra Marion Tinsley, el cual se retiró después de seis empates debido a problemas de salud. Aún así, un año más tarde, Chinook defendió su título contra Don Lafferty, ganando una partida y empatando las otras treinta y una. Después de esto, Jonathan Schaeffer, jefe del proyecto, decidió centrarse en intentar resolver las damas", es decir, buscar si era posible predecir el resultado de un partido (victoria, derrota, o empate), suponiendo que ambos jugadores jugasen a la perfección. Esta investigación concluyó en 2007 con la demostración de que el mejor resultado que un jugador podría obtener contra Chinook es un empate.

Aún así, no fue hasta la popularización de las máquinas arcade a finales de los 70 cuando se empezaron a desarrollar IA mucho con comportamiento mucho más complejos como en *Space Invader*, que disponía de un incremento en su velocidad según el jugador avanzaba niveles, o *Pac-Mac*, con una IA

capaz de dirigir a los fantasmas a través de un laberinto y que también ofrecía una IA para cada fantasma, otorgándoles un comportamiento diferente a cada uno[2].

En la década de 1990, la aparición de nuevos géneros llevó a la creación de nuevas IA capaces de manejar mayores cantidades de información, si bien, en algunos casos, como los juegos de estrategia en tiempo real (RTS), estas IA tenían severos problemas para manejar información incompleta, encontrar rutas y realizar decisiones en tiempo real, como puede verse en *Herzog Zwei*, considerado el primer RTS, o *Dune II*, que asentó los pilares del género. La IA también tuvo problemas con los juegos de disparos en primera persona (FPS). *Wolfenstein 3D*, primer juego del género, hacía uso de una Máquina de Estados Finita para manejar el comportamiento de los enemigos, esta IA, aunque avanzada para su tiempo, era incapaz de perseguir al jugador y se limitaba a ser una diana móvil capaz de devolver disparos. Esto cambió con *Doom*, desarrollado por el mismo equipo en 1993, que implementaba una IA que podía perseguir al jugador si este salía de su campo de visión, lo que, junto a un mapa más complejo, llevó a un desafío mayor para los jugadores.

Esta tendencia continuado durante las últimas décadas, dando lugar a ideas tan interesantes como *Creatures*, un simulador de vida artificial que usaba ANN para que las criaturas aprendieran e interactuaran con el entorno, o *Black & White*, que más allá de la IA enemiga, permitía tener una criatura que podías entrenar para que realizara las acciones que deseara el jugador mediante la Teoría del Condicionamiento Operante, es decir, premiándola cuando realizaba algo que se deseara y castigándola cuando hacia algo no deseable, pudiendo así obtener una criatura letal que luchara junto a el ejército aliado, o pacífica, capaz de ayudar a construir edificios u obtener recursos. Ambos juegos fueron una auténtica revolución al aplicar técnicas de Aprendizaje Máquina (ML), especialmente *Black & White*, cuyo trabajo, llevado a cabo por el investigador de IA Richard Evans y expuesto en Wexler (2002), con DT e incluso ANN, todavía no ha sido superado.

A pesar de esto, el avance de la IA en videojuegos ha continuado siendo uno de los pilares de la industria y, al mismo tiempo, uno de los mayores problemas que los desarrolladores encuentran. Esto se puede ver en muchos juegos de gran presupuesto como la saga Assassin's Creed, con una IA que se basa en patrones tanto a la hora de pelear como de reaccionar a situaciones en su entorno, *Skyrim* y los últimos *Fallout*, cuya IA solo es capaz de cargar contra el jugador, no sabe reaccionar a situaciones de sigilo e incluso se queda atascada con facilidad, o juegos de deporte como *Fifa*, con errores en el comportamiento de los jugadores que continúan iteración tras iteración de la saga.

Todo esto, en gran medida, se debe a que el comportamiento de los enemi-

[2] `https://web.archive.org/web/20121015061417/https://www.cnbc.com/id/41888021`

gos en un videojuego debe de ser predecible para ser divertido. Después de todo, si el enemigo tiene un comportamiento demasiado impredecible, puede resultar frustrante para el jugador, que no podrá aprender a enfrentarse a él.

Además, el coste de computación de una IA eficiente es considerable, añadido al hecho de que, en muchos casos, esta puede requerir de cálculos para todos los Personajes No Jugador (NPC) que se encuentren en el rango de visión (e incluso fuera del mismo), lo que representa un nivel de computación difícil de manejar incluso para computadoras de alta gama. Finalmente, el aumento en la popularidad de los juegos multijugador ha reducido la importancia de la inteligencia aliada y enemiga, ya que en el caso de los juegos Jugador contra Jugador (JcJ), la IA se ha vuelto una herramienta para practicar contra un enemigos más débil, mientras que en juegos cooperativos, la ausencia de una IA desafiante se ve balanceada por enemigos más fuertes y numerosos.

Aún así, las investigaciones de IA para NPC siguen avanzando, y es común encontrarse con enemigos con comportamientos complejos, así como IA capaces de analizar el estado del juego para ayudar/castigar al jugador cuando se encuentra en una situación demasiado difícil o cómoda. Un ejemplo de esto último es *Left 4 Dead 2* y su Director AI, que coloca objetos curativos cuando los jugadores están heridos y genera oleadas de enemigos que atacan a los jugadores que se encuentran solos para que se vean obligados a reunirse con su equipo, forzando la cooperación. Por otro lado *Rimworld*, un videojuego de supervivencia y construcción de bases, es otro ejemplo de una IA capaz de interferir externamente en el juego, generando eventos que favorecen al jugador cuando tiene colonos heridos o sin recursos, y enviando hordas de enemigos o provocando cortocircuitos cuando tienen buena salud y la capacidad de defenderse, ofreciendo así al jugador un desafío ajustado a la situación actual.

En su lugar, la mayoría de las investigaciones se centran en la IA no para crear comportamientos más complejos que mejoren la experiencia de juego, sino para hacer que esta IA sea capaz de jugar a estos videojuegos de la mejor forma posible.

AlphaStar, desarrollado por *DeepMind*, es un claro ejemplo de esto, se trata de una IA diseñada para jugar a *StarCraft 2* que fue capaz de superar al 99.8 % de los jugadores en un servidor oficial en las mismas condiciones que sus rivales (AlphaStar Team, 2019a). *DeepMind* logró esto usando distintas técnicas: Empezaron usando entrenamiento supervisado a partir de partidas de jugadores anónimos, lo que le ayudó a superar el 95 % de las partidas contra una IA con una habilidad moderada. Tras esto, usaron RL haciendo que los agentes (IA) se enfrentaran entre ellos. Esto permitió que los nuevos agentes, con un comportamiento más complejo, prevaleciesen sobre sus rivales dado que eran capaces de enfrentase las estrategias de sus rivales (agentes más antiguos). De este modo, se obtuvo el mejor agente, que

se enfrentó a jugadores profesionales y fue capaz de derrotar 5-0 a TLO, uno de los mejores jugadores del mundo (AlphaStar Team, 2019b).

2.1.3. Inteligencia Artificial aplicada al Desarrollo de Videojuegos

Con la invención de nuevas técnicas de IA, esta comenzó a ser aplicada al desarrollo de videojuegos no solo para el comportamiento de los aliados y enemigos, sino también para facilitar del trabajo de los desarrolladores.

Un ejemplo de esto es el *Style Transfer ML* de *Stadia*, desarrollado por Google, que permite cambiar el estilo gráfico de un juego con tan solo darle una imagen, aplicando las características de esa imagen a todos los elementos en pantalla mediante técnicas de ML[3].

Yannakakis (2012) explica otros usos de la IA que están empezando a surgir:

- Modelado de la Experiencia del Jugador (PEM): Usando técnicas de IA, es posible analizar la experiencia del jugador mientras juega para decidir aspectos que cambiar. Podemos dividir el PEM en tres tipos dependiendo cómo se obtiene la información. Pedersen et al. (2010)
  - PEM Subjetivo: La información es obtenida realizando encuestas a los jugadores sobre su experiencia con el juego u obteniendo retroalimentación de expertos.
  - PEM Objetivo: A través de técnicas de reconocimiento emocional y sensores fisiológicos, se obtiene información precisa acerca de lo que el jugador siente mientras juega. Por desgracia, esto es actualmente demasiado intrusivo y requiere equipamiento especializado, por lo que aunque ofrece mucha más información, solo puede ser obtenida realizando pruebas en entornos cerrados.
  - PEM Basado en la jugabilidad: Se recoge información sobre el comportamiento del jugador dentro del juego. Este es el método menos intrusivo, pero también ofrece información imprecisa.
- Generación de Contenido Procedural (PCG): Consiste en la creación de mapas, música, misiones y otros elementos de forma automática. Aunque esto se ha hecho durante décadas, usando IA es posible generar contenido adaptado al jugador y/o generar una cantidad casi infinita de elementos diferentes con una calidad similar a la que un humano podría lograr diseñar.
- Minado de Datos a escala masiva: En el caso de videojuegos con una gran cantidad de jugadores, es posible usar IA para analizar el comportamiento de los jugadores en el juego, como la estrategia de los

[3] `https://stadia.dev/blog/behind-the-scenes-with-stadias-style-transfer-ml/`

jugadores en *StarCraft* o los diferentes tipos de jugadores en *Tomb Raider: Underworld*, tal y como se muestra en Drachen et al. (2009).

El objetivo de este trabajo es aplicar un PEM Basado en la jugabilidad mediante IA par simular el estilo de juego de los jugadores y usar EC para probar múltiples estrategias de juego. De este modo, se puede obtener una retroalimentación de una gran cantidad de "jugadores"sin necesitar mostrar el juego al público o depender de *testers*.

2.1.4. Árbol de decisión

Un árbol de decisión (*Decision Tree* o DT) es un algoritmo de clasificación que consiste en un árbol que representa un algoritmo con una estructura *if-else*. Como otras estructuras de árboles, los DT están compuestos por nodos y aristas unidireccionales. Cada nodo representa el punto donde una decisión ha de ser tomada, mientras que las aristas representan las diferentes opciones para cada decisión.

Existen diferentes formas de crear un DT, dependiendo de la información disponible

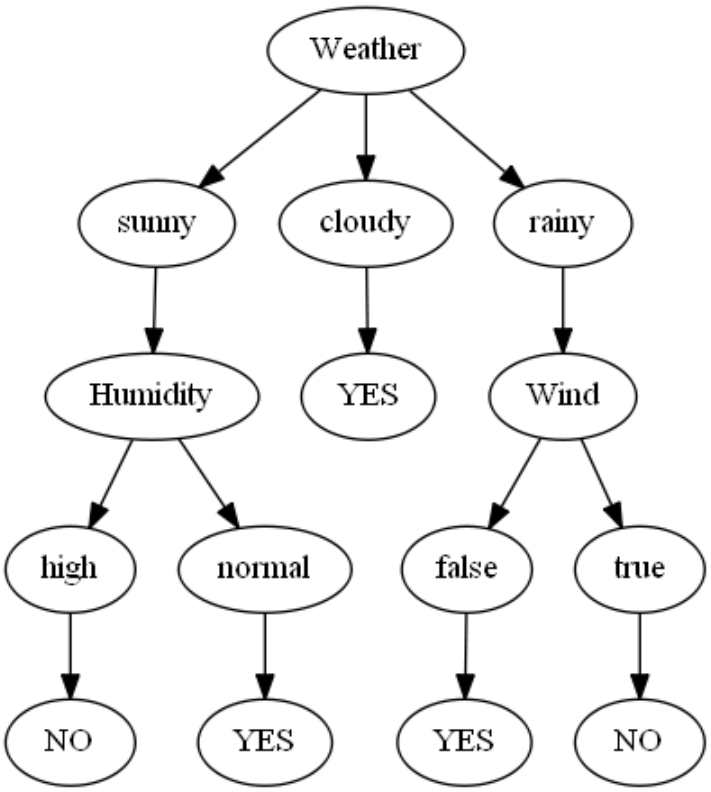

Figura 2.1: Ejemplo de un árbol de decisión.

```
public bool puedeJugar(string clima, string humedad, bool
    viento) {
    if(clima == "soleado") {
        if(humedad == "alta") {
            return false;
        } else {
            return true;
        }
    } else if(clima == "nublado") {
        return true;
    } else {
        if(viento) {
            return false;
        } else {
            return true;
        }
    }
}
```

Figura 2.2: Árbol de decisión como un algoritmo *if-else*.

En la figura Figura 2.1 se puede ver un ejemplo de un DT. Este ejemplo muestra un clasificador que, dado un conjunto de variables, decide si es posible jugar un partido de tenis. Los últimos nodos representan el resultado, si se juega el partido o no. Los nodos que empiezan con mayúscula representa la variable tomada en cuenta y los nodos en minúscula representan las diferentes opciones para esa variable. Este árbol podría ser perfectamente escrito como un algoritmo *if-else* tal y como se muestra en Figura 2.2.

2.1.5. Árbol de comportamiento

El árbol de comportmiento (*Behavior Tree* o BT) es otra técnica de IA basada en árboles. A diferencia de los DT, el BT no necesitan empezar a buscar la solución desde la raíz cada vez, sino que tiene hojas que representan tareas, que pueden tener éxito, fracasar o permanecer un tiempo en ejecución, y en ese caso continúan desde ese último nodo usado. Las tareas se ordenan mediante unos nodos de control de flujo que son principalmente el selector (similar a una puerta lógica O) y la secuencia (similar a una puerta lógica A).

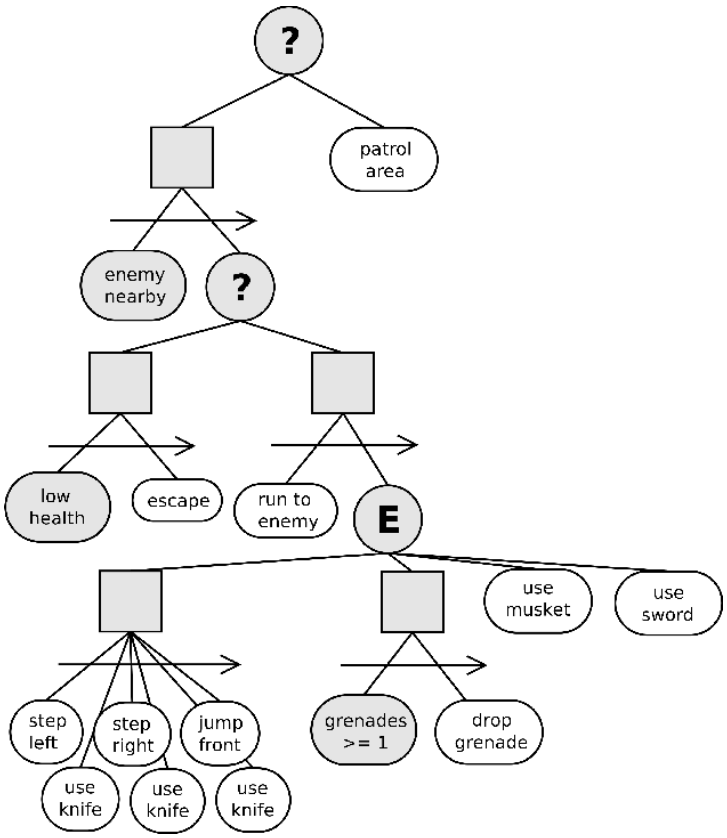

Figura 2.3: Ejemplo de un árbol de comportamiento.

2.2. Computación Evolutiva

La Computación Evolutiva (*Evolutionary Computation* o EC) es una rama de la IA inspirada por la Teoría de la Evolución con el objetivo de resolver problemas de optimización.

Emergió durante los años 60 con la Programación Evolutiva, el Algoritmo Genético y la Estrategia Evolutiva. Más adelante, en los 90, se añadió la Programación Genética. Estos cuatro algoritmos son considerados los pilares de los llamados Algoritmos Evolutivos (EA).

Un EA es un algoritmo de optimización basado en la Teoría de la Evolución. Estos algoritmos bio-inspirados simulan el proceso evolutivo de las poblaciones de seres vivos para encontrar el mejor individuo (solución óptima) de un problema dado. Aunque existen muchos EA, en una forma general, estos algoritmos comparten una serie de características:

- Las soluciones al problema son divididas en individuos. Cada individuo representa un conjunto de variables y tiene un valor que indica cómo de óptimo es el resultado obtenido.

- El algoritmo tiene una población, que es un conjunto de individuos, esta población cambia a través de de las generaciones, siendo una generación una iteración del algoritmo.
- Las poblaciones cambian gracias a la selección, cruce y/o mutación de los individuos.

El algoritmo funciona del siguiente modo:

1. Se inicializan los individuos dentro de la población (comúnmente de forma aleatoria o semi-aleatoria).
2. Se calcula el resultado de cada individuo y se le asigna un valor de *fitness*.
3. Selecciona los individuos que continúan. La selección normalmente favorece a los individuos con mejor *fitness*.
4. Se cruza a los individuos de la población, dando lugar a nuevos individuos (que pueden entrar en la población mediante diferentes métodos).
5. Los individuos mutan, es decir, reciben pequeños cambios en sus variables para ofrecer un nuevo resultado.
6. Mientras el algoritmo no considere que ha finalizado (ya sea por obtener un resultado óptimo o llegar al máximo de iteraciones), calcula el resultado de la nueva población y selecciona, cruza, y muta de nuevo hasta terminar.

Cada EA difiere principalmente en la forma en la que representa los individuos y los métodos usados para la selección, cruce, y/o mutación de dichos individuos. A partir de los primeros algoritmos, surgieron otros basados en modificaciones de los antiguos.

Además, el uso de EA se ha aplicado a otras técnicas de IA, dando lugar a técnicas híbridas como las ya mencionadas en Subsección 2.1.1. Entre estas técnicas, además de las Redes Neuronales Evolutivas, se encuentra el Árbol de Búsqueda de Monte Carlo Evolutivo. En general, EC contienen todas las técnicas que usan EA. Debido a esto y al auge de la EC, actualmente hay numerosos métodos y campos de estudio. Entre estas técnicas, algunas de las más interesantes son:

- **Redes Neuronales Evolutivas (Neuroevolución):** Haciendo uso de EA, la Neuroevolución es capaz de entrenar ANN sin necesidad de ejemplos de entrenamiento. Con el aumento de la capacidad de computación en los últimos años, esta rama está realizando grandes avances, como puede verse en las investigaciones de (Kenneth O., 2017) (*Uber*).

Este enfoque es especialmente interesante para la creación de IA capaces de jugar videojuegos diferentes, considerando que solo requiere un marcador de puntación para comprobar si está jugando bien o mal, sin necesidad de ejemplos.

- **Árbol de Búsqueda de Monte Carlo Evolutivo (Evolutionary MCTS):** Una aproximación del MCTS haciendo uso de EA. Como se muestra en Lucas et al. (2014), el EA ayuda a incrementar la eficiencia y la velocidad de generación del árbol.

- **Vida Artificial (AL):** Se centra en la creación de sistemas capaces de comportarse como si fueran seres vivos. Esta se divide en *soft*: Programas de software (normalmente simulaciones), *hard*: Hardware (generalmente robots) con un comportamiento "vivo", y *wet*: Centrado en la investigación molecular para sintetizar vida. En Aguilar et al. (2014), se explican con mucho más detalle la AL, su presente y futuro.

- **Inteligencia Enjambre (SI):** Inspirada en colonias de hormigas o abejas, presenta un algoritmo en el que los individuos, en lugar de reproducirse y mutar, interactúan con los otros para alcanzar un punto óptimo. Hay otros algoritmos bioinspirados que funcionan de una forma similar, como el inspirado en murciélagos expuesto por Yang (2010).

- **Algoritmo Cultural (CA):** Considerado una extensión del EA, hace uso de un .espacio de creencias", que altera los genomas de los individuos, de modo que tienden a converger, mientras que al mismo tiempo, los genomas de los mejores individuos modifican el espacio de creencias.

Como ya se ha mencionado, estas técnicas hacen uso de EA. A continuación, comentaremos con más detalle el funcionamiento de estos algoritmos.

2.2.1. Algoritmo Genético

Las variables de los individuos se representan mediante cadenas binarias.

Durante el cruce, se simula la meiosis, seleccionando dos individuos, dividiendo las cadenas en dos, y cruzándolas entre ellas. Para la mutación se seleccionan bits y se cambia sus valores.

Tal y como se expone en Thengade y Dondal (2012), los Algoritmos Genéticos (AG) son útiles en cuanto a optimización de funciones, especialmente aquellas no derivables o demasiado complejas de derivar. Sin embargo, no son eficientes con problemas que requieren la representación de variables más complejas (no numéricas) como *DT*.

2.2.2. Estrategia Evolutiva

La EE se usa normalmente con problemas que requieren una representación de vectores de números reales. La principal diferencia respecto a otros algoritmos es el método de mutación, que usa una Distribución Normal Multivariante para mutar a los individuos.

Una variación de estos algoritmos es la Estrategia Evolutiva de Matriz de Adaptación Covariante (*CMA-ES*), que usa una Matriz de Covarianza dentro de la Distribución Normal. Tal y como expone Beyer (2007) esta matriz se actualiza a través de las generaciones. Con esto, los individuos tienden a mutar al óptimo mucho más rápido, haciendo esta EE la más usada y uno de los mejores algoritmos de optimización en cuanto a trabajar con números reales en funciones de muchas dimensiones.

Otra variación de la EE son las Estrategias Evolutivas Naturales. *wierstra2011natural* muestra que este algoritmo es capaz de superar el rendimiento del EE tradicional gracias al uso de una gradiente natural.

2.2.3. Programación Evolutiva

Similar al AG, cambia la representación de las variables de los individuos dependiendo del problema a resolver. Además carece de método de mutación, de modo que la evolución se produce mediante el cruce de los individuos (Fogel et al. (2011)).

Como comenta Bäck et al. (1997), la PE y la EE son dos tipos de algoritmo que se basan en el uso de mutaciones de distribución normal aleatoria como método principal de optimización. Sin embargo, hay una diferencia esencial entre estos algoritmos, la ausencia de método de reproducción en la PE, además de un método de selección más permisivo. Esto hace que el rendimiento de la PE sea peor generalmente, aunque como se muestra en Wong y Guan (2000), la PE tiene usos realmente interesantes y prometedores como la restauración de imágenes.

2.2.4. Programación Genética

Es una especialización del AG donde cada individuo es un programa (un algoritmo en si mismo) representado como un árbol.

Este algoritmo es especialmente interesante en el contexto de este trabajo, ya que permite la creación de estrategias de juego automáticas, haciendo posible generar IA y comparar su eficiencia para obtener otras mejores.

Puede usarse en modelos predictivos y de reconocimiento.

Un acercamiento de estos algoritmos a los videojuegos es la Programación Genética de Terreno, como la expuesta en Frade et al. (2009), que consiste en el uso de PG para crear terreno automáticamente.

2.2.5. Computación Evolutiva en videojuegos

En los últimos años, la CE ha ganado fuerza en el campo de los videojuegos, tanto en investigación en IA como en *testing*. Esto es debido a que la CE permite testear todas las posibilidades que ofrece un videojuego sin necesidad de ejemplos, lo que hace que resulte fácil trabajar con videojuegos que en otros casos requerirían ejemplos de cada posible estado del juego.

Tal y como se propone en Baier y Cowling (2018), es posible usar CE para crear un *MCST* que permita a la IA jugar videojuegos JcJ con turnos de múltiples acciones. La complejidad de estos juegos radica en el hecho de que las decisiones no pueden ser tomadas en base a la mejor acción actual, ya que la mejor decisión puede ser un conjunto de acciones que por si solas son la mejor y no serían tomadas en cuenta.

A través de la Neuroevolución, Miranda et al. (2016) muestra que es posible entrenar una ANN mediante CE para crear ejemplos que de otro modo requerían cientos de partidas de ejemplo. Por desgracia, tal y como se expone en el artículo, el algoritmo no dispone de una gran precisión todavía, pero muestra que es un enfoque viable que puede funcionar en el futuro.

Otro ejemplo es el uso de EA para la creación de una IA que juegue al *Hearthstone*, como se muestra en García-Sánchez et al. (2020). Esto se hace gracias a una función de *fitness* que calcula la diferencia entre el anterior y actual estado del juego después de una acción. Para gestionar el comportamiento, la función tiene veintiuna variables de peso, cada una ligada a una o varias diferencias de estado, que son las variables que conforman el genoma de los individuos.

Relacionado con esto, también se ha demostrado que es posible usar EA para testear barajas de *Hearthstone* para encontrar las mejores combinaciones [García-Sánchez et al. (2018)]. Aunque en este caso los resultados son prometedores, con un gran porcentaje de victorias, tal y como se comenta en el propio artículo, solo se entrenó y probó contra una IA predefinida, por lo que estas barajas pueden no ser óptimas contra jugadores reales.

2.3. Defensa de torres

El género de Defensa de torres (en inglés *Tower Defense*) surgió en 1990 con *Rampart*, de *Atari Games*, en este videojuego el objetivo es construir un fuerte y colocar cañones para defendernos de oleadas de barcos que intentan atacar nuestro castillo. En el caso de que sobrevivamos al ataque, habrá que reparar las defensas y colocar más cañones. Esta mecánicas, si bien simples, fueron las bases de lo que más adelante serían los *Tower Defense*.

Aún así, el género pasó desapercibido durante años, con algunos minijuegos dentro de videojuegos como *Final Fantasy VI* y *Final Fantasy VII*, que aunque se encontraban leos de los estándares actuales del género, se podrían

considerar los primeros acercamientos al mismo.

Esto cambió con *Warcraft 3*, de *Blizzard*. Gracias al editor de mapas que incorporaba este juego, los jugadores podían crear sus propios niveles. Con esto, el género resurgió con mapas creados por otros jugadores donde había que defenderse frente a oleadas de enemigos que recorrían un camino prefijado colocando unidades en los laterales del camino para que atacaran automáticamente a estos. Esto podrían considerarse los cimientos de lo que sería el género.

A principios de 2007, con el surgir de los juegos de móviles y los videojuegos web creados con Adobe Flash, el género finalmente emergió, con docenas de juegos de inmensa popularidad con *GemCraft* o *Desktop Tower Defense*. El éxito de estos videojuegos no fue ignorado y, con la aparición de los móviles con pantallas táctiles, los desarrolladores adaptaron estos juegos al mercado móvil con gran éxito, aunque uno de los mayores exponentes del género, *Plants vs Zombies*, fue creado inicialmente como videojuego de PC, adaptado posteriormente a numerosas plataformas.

En los últimos años el género no ha perdido popularidad y ha continuado teniendo un gran éxito, especialmente en el mercado móvil. Además, esta popularidad ha propiciado nuevas variantes, como la combinación del *Tower Defense* con las cartas coleccionables dando lugar a juegos como *Clash Royale*. Este nuevo género se centra en el combate JcJ, al igual que los juegos de cartas, con barajas creadas por los propios jugadores. La diferencia respecto a los juegos de cartas es que no existen turnos y los jugadores no seleccionan los objetivos de sus unidades, sino que las "sueltan.[en] el mapa, generalmente compuesto de 2 o más caminos, y las unidades se desplazan por los mismos y atacan a los enemigos que encuentran a su paso automáticamente. El objetivo de la partida es hacer que las unidades lleguen a la base del rival, generalmente teniendo que destruir una fortaleza o avatar, que dispone de una gran cantidad de vida y, cuando el jugador la destruye, gana la partida.

Al igual que en los videojuegos de cartas, el jugador dispone de una baraja formada por diferentes cartas (si bien estas suelen ser mucho más pequeñas que las barajas de los TCG tradicionales) que aparecen aleatoriamente según el jugador usa las que se encuentran en su mano y, pero a diferencia de los TCG, las barajas no se vacían según se juegan cartas, sino que esta indica las posibles cartas que pueden aparecer en la mano del jugador de forma ilimitada. Además, también existe un sistema de "maná", necesario para usar las cartas, que se regenera automáticamente según pasa el tiempo. Esto hace que el jugador tenga que esperar una cantidad concreta de segundos para usar una carta, teniendo que esperar más tiempo si quieren colocar unidades que requieren usar más maná, dando lugar a diferentes estrategias.

2.3.1. Warrior Defense

Para probar la herramienta, usaremos un videojuego de defensa de torres llamado *Warrior Defense*. Se trata de un videojuego de código libre creado en *Unity* y disponible en el sitio web Unity List`https://unitylist.com/p/12p6/warrior-defense-factory`. Fue creado como demostración del patrón Factoría en *Unity*. Aunque es un juego incompleto, se puede usar libremente, tanto los recursos como el código, y es lo bastante simple como para modificarlo fácilmente y adaptarlo a nuestras necesidades.

Originalmente, *Warrior Defense* es un juego que consiste en dos jugadores que invocan criaturas en una arena para pelear, sin embargo no dispone de ningún tipo de puntuación u objetivo. Para poder usar este juego para nuestra herramienta, es necesario dar al jugador una meta y una forma de evaluar su rendimiento. Para esto, se modifica el juego creando un *Tower Defense* en 2D con mecánicas TCG, similar a *Clash Royale*. El juego consiste en dos jugadores, uno en cada lado de la pantalla, que cuentan con una baraja de unidades que pueden invocar usando maná.

Figura 2.4: Captura del Warrior Defense

Los principales componentes del juego son:

- Unidad: Las unidades invocadas se mueven siempre hacia delante, intentando alcanzar al jugador enemigo. Si encuentran una invocación enemiga en su camino, intentarán luchar contra ella y, si la derrotan, seguirán moviéndose hacia el jugador enemigo. Las diferentes unidades son:
  - *Magic*: Reaper (Melee), BatCat (Melee), Wizard (Ranged), FlyingEye (Ranged).
  - *Melee*: HeroKnight, MartialHero, Mushroom, Skeleton.
  - *Ranged*: Archer, Cassiopeia, Goblin, DemonSlug.

La principal diferencia entre esas unidades es su rango de ataque y sus atributos, aun así todos ellos se comportan de la misma forma.

- Puntuación: Los jugadores obtienen puntos cuando sus unidades llegan al otro extremo del campo de batalla.
- Maná: El maná es usado para invocar unidades. Se recarga una cantidad fija por segundo. Las mejores unidades cuestan más maná, por lo que requieren esperar más tiempo para ser lanzadas. La cantidad máxima que un jugador puede tener es 1000.
- Unidades disponibles: Cada jugador tiene una baraja con las unidades que puede invocar. A diferencia de los juegos TCG, estas unidades están prefijadas y no cambian con el paso del tiempo o según se invocan las unidades.

La función de *fitness*, en este caso, es la puntuación obtenida. Para ello se resta la puntuación del rival a la del jugador, buscando maximizar el valor. De este modo, forzamos que el algoritmo busque individuos capaces de obtener más puntos, es decir, el jugador que es capaz de invocar unidades que atraviesen las lineas enemigas y lleguen a la meta.

2.4. Estrategia en tiempo real

Los videojuegos de estrategia en tiempo real o RTS (siglas en inglés de *Real-Time Strategy*) son videojuegos de estrategia en los que no hay turnos sino que el tiempo transcurre de forma continua para los jugadores.

Los videojuegos en tiempo real son uno de los subgéneros de los juegos de estrategia más dinámicos que hay.

Los RTS están pensados para ser jugados de forma muy dinámica y rápida. A diferencia de los basados en turnos no precisan un planteamiento tan pausado de las decisiones y se centran muy a menudo en la acción militar. La recolección de recursos suele ser simple, solo hay materias primas. Las batallas se representan a una escala de refriega, aunque hay varios juegos que se centran en representar batallas multitudinarias con millares de unidades en el terreno, como sucede en las batallas de la saga *Total War* o en el legendario *StarCraft*.

Las partidas en los RTS se desarrollan en un mapa, compuesto por unidades, edificaciones, recursos y terreno.

- Unidades: Son individuos o grupos que forman una unidad independiente, estos son controlados por el jugador a través de la interfaz del juego. El numero máximo de unidades depende del juego pero normalmente lo manejan a través de poblaciones, pudiendo variar el numero gracias a diferentes edificaciones. Se suelen dividir en unidades civiles o

militares. Las civiles suelen encargarse de la recolección de materiales o construcción de edificaciones. Las militares en cambio su funcionalidad es únicamente la batalla y el asedio en contra de los enemigos. Todas las unidades suelen tener estadísticas propias como vida,ataque y defensa, aunque en ciertos juegos estas se amplían.

- Edificaciones: Son objetos normalmente inamovibles y construidos por las unidades civiles. Se suelen dividir en tres categorías principalmente:
  - Producción: Encargadas de la producción de unidades a cambio de recursos.
  - Recolección: Sirven como punto de destino para las unidades encargadas de la recolección de recursos para depositar los recursos, o como almacén de estos aumentando la cantidad máxima posible.
  - Defensa: Su función principal es defender de las unidades enemigas que intenten atacar la base del jugador.

  No tienen que pertenecer a una sola categoría por obligación, al igual que dependiendo del juego se pueden ampliar el número de categorías.
- Recursos: suelen estar distribuidos por el mapa y pueden ser tanto finitos como infinitos.
- Terreno: El terreno en el que se pueden mover las unidades. Este proporciona camino posibles a seguir de todas las unidades y restricciones a otras. Por ejemplo un río no puede ser atravesado si la unidad no tiene la capacidad para nadar o volar.

2.4.1. MicroRTS

MicroRTS es el prototipo de otro juego que usaremos para probar nuestra herramienta. Este juego fue proporcionado por uno de nuestros directores, siendo la base para una de las práctica de la asignatura de Inteligencia Artificial para Videojuegos (IAV) de 3º del Grado en Desarrollo de Videojuegos.

MicroRTS pertenece al genero de los RTS y aunque no se pueda considerar un videojuego, ni sea muy extenso en funcionalidad, es perfecto para demostrar la utilidad de nuestra herramienta.

La IA base sobre la que se aplicará nuestra herramienta no está definida originalmente. Por lo tanto hemos creado una IA sencilla pero que permite a nuestra herramienta cambiar por completo su comportamiento, demostrando así los amplios límites de optimización que puede alcanzar.

El juego trata de la destrucción de la base contraria a través del uso de unidades. De las cuales hay tres tipos:

- Recolectoras: Recogen los recursos dispuestos en las zonas de recursos.

Figura 2.5: Captura del MicroRTS

- Exploradoras: Unidades de combate rápidas y débiles.
- Destructoras: Unidades de combate fuertes pero muy caras, cuyo límite máximo es reducido.

Los tipos de instalaciones de cada bando son 2, la base principal desde la que se generan las unidades y la instalación de procesamiento donde se almacenan todos los recursos.

En el mapa se pueden encontrar puntos de recolección, torres y pequeñas bases.

- Puntos de recolección: En ellos se encuentran los recursos a recoger.
- Torres y pequeñas bases: No pertenecen a ninguno de los jugadores, y en concreto las torres atacan a cualquier unidad que se acerca a ellas.

La función de *fitness* que evaluará que tan bien se han desenvuelto, se dará por las unidades destruidas en cada bando y el tiempo que han tardado en destruir la base enemiga o en ser destruidos.

2.5. Unity

Unity es un motor de videojuego desarrollado por *Unity Technologies* disponible para Windows, Linux, y Mac OS. Permite desarrollar videojuegos multiplataforma y es de licencia gratuita siempre y cuando los ingresos obtenidos mediante su uso sean inferiores a 100.000$ en el último año.Un motor de videojuegos esta diseñado para desarrollar videojuegos desde su núcleo, incluyendo código, gráficos,físicas, etc.

Una de las principales ventajas de este motor frente a la competencia es su facilidad de uso y el hecho de que los *scripts* se crean en C#, un lenguaje más sencillo de aprender en comparación con otros como C++, usado por *Unreal Engine*. Si bien uno de los mayores contras de este motor es su rendimiento, debido en gran parte a la gestión de la memoria y las librerías .Net, lo que hace que también sea más complejo desarrollar juegos con gráficos punteros, esto no ha impedido que sea utilizado por muchos desarrolladores, especialmente de la escena *indie*, donde los gráficos no son tan exigentes.

Entre los juegos de mayor éxito creados mediante *Unity* destacan *Kerbal Space Program, Firewatch, Hollow Knight* y *Cuphead* así como otros juegos de mayor presupuesto como *Ori and the Blind Forest* o incluso *Hearthstone*, de *Blizzard*.

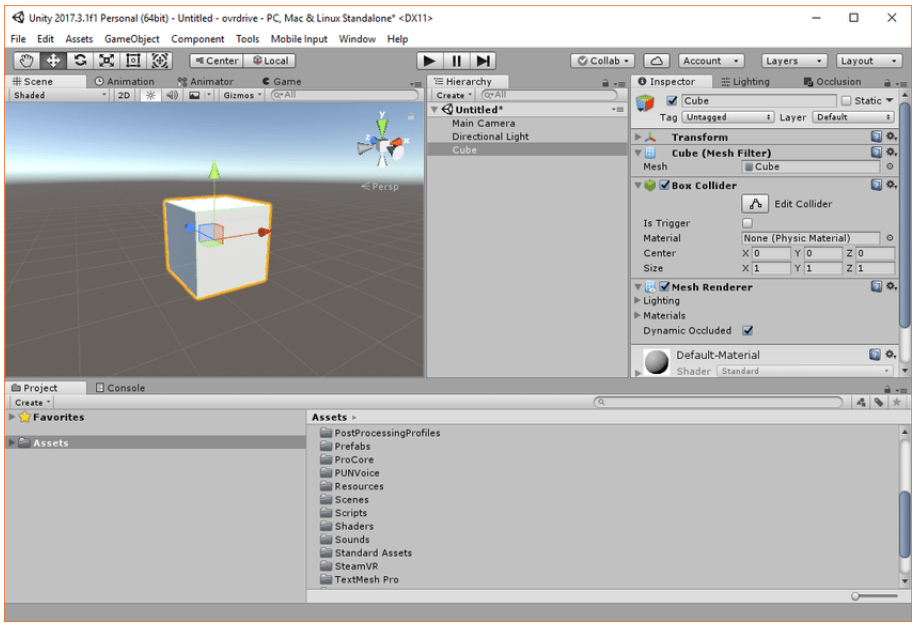

Figura 2.6: Interfaz de Unity.

En 2.6 podemos ver la interfaz de *Unity*. Aunque las ventanas y su visualización pueden ser diferentes dependiendo de cómo las coloque el usuario.

Las principales ventanas de *Unity* están explicadas debajo:

- *Inspector*: Muestra la información del objeto seleccionado. Aquí podemos modificarle, su posición en la escena, las variables de sus *scripts* asociados, etc.

- *Scene*: Permite el libre movimiento de los objetos. Aquí se puede crear interfaz de usuario (UI), crear el mapa, colocar los objetos, etc.

- *Hierarchy*: Los objetos están ordenados aquí, podemos seleccionar fácilmente los objetos de la jerarquía, especialmente los que no tienen representación gráfica y no pueden ser seleccionados en la Escena.

- *Game*: Muestra el videojuego tal y como lo vería el usuario, es decir, el juego con la interfaz de usuario y sólo viendo la cámara activa.

Unity, como muchos otros motores de videojuegos, permite la creación de *plug-ins* o herramientas que ayudan a los desarrolladores a crear videojuegos. Los *plug-ins* pueden ser creados de forma nativa utilizando lenguajes basados en C como C, C++ u Objective-C, esto nos permite tener acceso a las llamadas del sistema operativo. También es posible desarrollar *plug-ins* utilizando código .Net, que se denominan *Managed plug-ins*, pero sólo tienen acceso a las características que soportan las librerías .Net. Por supuesto, lenguajes como C# o Python pueden ser utilizados a través de intérpretes para C/C++. Estas herramientas ayudan a los desarrolladores dando soporte a diferentes controladores, creando terrenos complejos, animaciones o física.

En nuestro caso, el Algoritmo Evolutivo no requiere C++ o .Net porque se puede hacer como si fuera un *script* externo creado por el desarrollador del videojuego. Esto facilita la creación de la herramienta y no afecta al rendimiento porque el algoritmo no requiere ningún acceso especial.

Unity está diseñado para ser fácil de usar, por lo que los desarrolladores no necesitan mucho esfuerzo ni conocimientos para crear sus videojuegos. Aunque no es necesario entender conceptos sobre campos como los gráficos o la física ya que no están relacionados con este trabajo, es importante saber que el motor tiene un simulador de física con detección de colisiones, variables de gravedad y masa, etc, e incluso un sistema de partículas. También es posible crear animaciones, tanto en 2D como en 3D, asignando el cambio de animación a diferentes variables. Tiene también ya creado un sistema de escenas para poder dividir fácilmente el juego en diferentes partes para no tener que realizar una carga de objetos extremadamente grande.

Para hacerlo más fácil de usar para nuevos desarrolladores utilizan una estructura de componentes orientada a objetos, en el cual los *scripts* creados por el desarrollador se asocian a los objetos y amplían su funcionalidad directamente. Esto no impide la creación de *scripts* que no se asocien a objetos directamente si el desarrollador no quiere que dependan de un objeto para su funcionamiento.

Todas las variables dentro del motor pueden ser cambiadas en la interfaz de *Unity*, pero es posible cambiarlas a través de *scripts*. Normalmente, los *scripts* son la forma de cambiar las variables (excepto las fijas, establecidas a través del inspector o de los *scripts* al inicio de la ejecución).

El desarrollo de un videojuego se divide en múltiples fases durante el desarrollo y se pueden hacer todas o la mayoría en *Unity*,aunque tiene más presencia en unas que otras:

- Empezando por el prototipo tanto de la idea básica del juego como de las mecánicas, esto *Unity* hace que sea bastante sencillo al tener modelos sencillos básicos y componentes de físicas tanto 2D como 3D.

- Creación de las mecánicas y niveles, para esta fase *Unity* tiene muchas formas de agilizar el proceso, sobretodo en 2D con la posibilidad de crear mapas basados en *tiles* (división del mapa en cuadrados) que se pueden rellenar en el editor y la posibilidad de recortar *sprites*. Al igual que la facilidad para mover objetos y colocarlos en la escena sin iniciar el juego. Para la creación de inteligencias artificiales si se incluyesen al juego tiene múltiples herramientas, aunque estas ya son de pago normalmente para facilitar su creación Como puede ser el caso de arboles de comportamiento.

- Testeo. Durante esta fase se probaría el juego una y otra vez, este proceso se hace primero por parte de los desarrolladores únicamente, normalmente se prueba de esta manera tras meter una mecánica nueva o cada cierto tiempo. Hay pocas herramientas en *Unity* para esta parte, y aun menos para inteligencia artificial. Suele ser prueba y error hasta encontrar donde está el problema y solucionarlo.

- Equilibrado: esta fase suele realizarse tras terminar escenas o gran parte del juego.Trata únicamente del evaluado de las mecánicas y juego a base de pruebas reiteradas tanto con usuarios como por los desarrolladores. Para ciertos ámbitos pueden existir herramientas automatizadas que ayuden con el proceso. Curiosamente en *Unity* hay muy pocas o ninguna en el ámbito de la inteligencia artificial que es lo que buscamos solucionar y ampliar con este trabajo.

- Comercialización: *Unity* aporta varias herramientas o formas de ayuda para remunerar tu juego, sobretodo con publicidad. También facilita mucho la creación de instaladores del juego.

Capítulo 3

Objetivos y especificaciones

Para iniciar el desarrollo de la herramienta es necesario plantear unos objetivos, que posteriormente se transformarán en especificaciones, y facilitarán el diseño y la implementación de esta.

En este capítulo se detallan los objetivos considerados necesarios para que la herramienta sea apta. A continuación, se muestra la especificación de requisitos de esta herramienta.

3.1. Objetivos

Los objetivos de la herramienta han sido planteados desde el punto de vista de un desarrollador con nociones muy básicas de programación e IA, y conocimientos relativamente avanzados de desarrollo de videojuegos y *Unity*.

Esto se ha realizado en base a los conocimientos en desarrollo de videojuegos del equipo. Dado que no se ha podido contar con desarrolladores expertos, estas especificaciones probablemente se puedan mejorar. Sin embargo, se ha intentado generalizar lo máximo posible enfocándose en desarrolladores con poco presupuesto y sin conocimientos de programación.

El principal objetivo de la herramienta es generar una IA sin necesidad de ejemplos de entrenamiento. Sin embargo, dado que la herramienta está orientada a usuarios sin experiencia en programación, es necesario añadir una serie de objetivos que faciliten su usabilidad. Además, ya que el algoritmo evolutivo se ofrece a utilizar diferentes métodos de selección, reproducción y mutación, así como distintos modos de representar los individuos, es una buena idea que la herramienta sea lo más modular posible para que se puedan añadir nuevos componentes en cualquier momento.

Por todo esto, los objetivos de la herramienta de equilibrado para *Unity* son los siguientes:

- Generar una IA capaz de enfrentarse a una IA básica previamente creada.

- Tiene que ser lo más genérica posible, es decir, debe de poder usarse en cualquier videojuego siempre que este se encuentre bien estructurado.
- Debe de ser fácil de usar. Con tan solo unos conocimientos muy básicos de algoritmos evolutivos debería ser posible ejecutar la herramienta y obtener un resultado.
- Todo debe de poder ser manejable desde la interfaz, sin modificar código.
- Añadir nuevos componentes como individuos, genes o métodos de selección, reproducción y mutación debe de resultar sencillo y sin necesidad de modificar el código interno de la herramienta.
- Generar árboles de decisión simples que se puedan adaptar a cualquier videojuego.

3.2. Especificación

A partir de los objetivos que se mencionan arriba, así como el contexto de la herramienta, se puede generar la especificación de la misma.

3.2.1. Restricciones

Teniendo en cuenta el entorno de la herramienta, es necesario tener en mente unas restricciones que limitarán su implementación. Estas restricciones vienen dadas principalmente por el software hacia el que está orientado la herramienta (*Unity*).

- Lenguaje de programación: Dado que se trata de una herramienta par *Unity*, es obligatorio utilizar C# para el desarrollo.
- El sistema no usará procesamiento multihilo. *Unity* no tiene soporte para procesamiento multihilo, por lo que plantear una arquitectura con multihilos sería complejo y daría lugar a errores.
- El sistema no hará comprobaciones ni suposiciones sobre la existencia o el tipo de los objetos. Si durante la ejecución no existe un objeto o este no pertenece a la clase o tipo esperado, se finalizará la ejecución del algoritmo y se mostrará un mensaje de error.
- A pesar del anterior punto, el sistema debe de ser capaz de trabajar con cualquier tipo de valor dentro de la parte genérica del mismo.
- No se hará uso de ninguna base de datos para almacenar la información.

3.2.2. Características del usuario

El usuario siempre será (o debería ser) un desarrollador de videojuegos con unos conocimientos avanzados de *Unity*, además, también debería tener nociones básicas acerca del funcionamiento de un algoritmo evolutivo. Sin embargo, no es necesario que tenga conocimientos avanzados de programación.

3.2.3. Historias de usuario

Ya que el proyecto usa una metodología ágil y la naturaleza de la herramienta hace que resulte muy complejo detallar su flujo de datos, se ha optado por omitir los Casos de Uso y utilizar Historias de Usuario, que si bien son más genéricas, también se adaptan mejor al concepto de centrarnos en el usuario, dejando más abierta la implementación, que puede variar según se progrese en el desarrollo y se encuentren obstáculos.

- Como usuario, quiero elegir el objeto o conjunto de objetos en la escena para decidir cuáles quiero replicar y evaluar y cuáles no es necesario.
- Como usuario, quiero elegir el número de individuos para ajustar el rendimiento del algoritmo en base a mi capacidad de computación.
- Como usuario, quiero seleccionar el número de generaciones y el tiempo de cada una para decidir cuánto tiempo quiero ejecutar el algoritmo hasta obtener los resultados.
- Como usuario, quiero elegir el modo de representar la información del individuo para adaptarlo a la IA de mi videojuego.
- Como usuario, quiero elegir los métodos de selección, reproducción y mutación para usar los métodos que mejor se adapten a mi videojuego.
- Como usuario, quiero elegir la función de *fitness* para decidir la forma de evaluar a mis individuos.
- Como usuario, quiero obtener unos resultados tras la ejecución del algoritmo para poder introducirlo en la IA de mi videojuego.
- Como usuario, quiero elegir las probabilidades de mutación y reproducción para ajustar el algoritmo a las necesidades de mi videojuego.
- Como usuario, quiero crear mis propios métodos de selección, reproducción y mutación, y usarlos dentro del algoritmo para personalizar al máximo la ejecución del mismo.
- Como usuario, quiero elegir los posibles valores de los genes para que representen de forma realista el comportamiento de mi IA.

- Como usuario, quiero ver el tiempo de ejecución de la herramienta para saber cuánto falta para finalizar la ejecución.
- Como usuario, quiero elegir la velocidad del juego para reducir el tiempo de ejecución del algoritmo sin perder rendimiento.

3.2.4. Funciones

A partir de las historias de usuario se han creado una serie de funciones básicas y necesarias de la herramienta. La ejecución del algoritmo evolutivo se ha omitido dado que es algo intrínseco de la herramienta.

- Seleccionar *prefab* del individuo.
- Elegir métodos de selección, reproducción y mutación y sus probabilidades.
- Ajustar parámetros del algoritmo (número de individuos, número de generaciones y elitismo).
- Cambiar la velocidad del videojuego.
- Seleccionar el tiempo de ejecución para calcular el *fitness* de los individuos de cada generación.
- Añadir observador de las puntuaciones del juego.
- Elegir individuo y genes a utilizar en el algoritmo.

Capítulo 4

Metodologia

Durante la realización del proyecto se siguió una metodología similar a *SCRUM*, en la cual se dividió el tiempo total disponible para el TFG en tres hitos.

En el primer (23 de diciembre de 2020) y segundo hito (25 de marzo de 2021) se presentaron los avances del proyecto a los miembros de *Narratech*, el grupo de investigación al cuál pertenece nuestro director de proyecto, en formato de presentación como práctica de cara a la presentación final del proyecto. El tercer (1 de junio de 2021) y último hito será la entrega del proyecto.

Para poder mantener un trabajo constante se realizaron *sprints* con duración de dos semanas, después de las cuales habría una reunión, en esta reunión se hablaba de los progresos realizados en el *sprint* anterior, problemas o cuestiones que surgían durante el desarrollo y se organizaba el trabajo de cara al siguiente *sprint*.

4.1. Herramientas

A continuación vamos a describir las herramientas utilizadas durante el desarrollo de nuestro proyecto.

4.1.1. Lenguaje de programación

Se utilizó el lenguaje de programación orientado a objetos C# como lenguaje principal, debido a que es el lenguaje nativo de *Unity*. De esta manera se evitaría la necesidad de crear o utilizar herramientas externas para *Unity*.

4.1.2. Entornos de desarrollo

Las herramientas que hemos usado para trabajar y escribir código son *Unity* y *Visual Studio*.

- *Unity* es la plataforma y motor destino para el trabajo. Cabe destacar que todos los miembros del equipo poseen cierta experiencia utilizando esta tecnología para videojuegos.

- *Visual Studio* es el entorno de desarrollo integrado por defecto de *Unity*, permite hacer pruebas de *debug* mas fácilmente, además de haber sido el editor de código más utilizado durante todo el grado para las diferentes materias.

4.1.3. Herramientas de edición de texto

- *Overleaf* es un editor colaborativo de *LaTeX* donde hemos realizado el desarrollo de la memoria del TFG. Seleccionamos este editor en específico por recomendación de nuestro tutor, ya que es muy utilizado por estudiantes de grado que realizan sus proyectos y nos ofrece un entorno donde compartir el avance de la memoria del proyecto en tiempo real.

- *Google docs* es el editor de texto que utilizamos para llevar las actas de las reuniones realizadas al terminar cada *sprint*, además de los resultados de investigaciones de cara a estas reuniones.

4.1.4. Herramientas de comunicación

- *Slack* ha sido la herramienta de mensajería directa utilizada para comunicación general entre el equipo y el tutor. El motivo principal se debe a que *Narratech* realiza sus comunicaciones en esta herramienta, por lo que nos da acceso a los comunicados con destino a todos los grupos que realicen su TFG, TFM y doctorado con *Narratech*.

- *Discord* y *whatsapp* como servicios de mensajería instantánea que utilizamos a diario para comunicarnos entre los miembros del equipo.

- *Google meet* fue el servicio de videollamada elegido para realizar las reuniones de cada *sprint*, además de las reuniones de los hitos donde se presentaron los avances del proyecto a los demás miembros de *Narratech*.

- *Google calendar* servicio de calendario para mantenernos al día con las reuniones programadas para los *sprints* e hitos.

4.1.5. Herramientas de almacenamiento y control de versiones

- *GitHub* es la herramienta de control de versiones elegida para almacenar todos los ficheros del proyecto de *Unity*. Decidimos utilizar *GitHub* sobre otras herramientas similares debido a que el equipo contaba con experiencia sobre su uso, lo que nos daba más seguridad sobre su fiabilidad.

- *Google Drive* fue utilizado como servicio de almacenamiento para tener un acceso rápido a los ficheros externos al código, como actas de reuniones, presentaciones que se han ido preparando, investigaciones sobre información necesaria para el desarrollo del proyecto, etc.

Capítulo 5

Desarrollo de la herramienta *EvoUnity*

Al igual que con cualquier otro software, para crear la herramienta es importante plantear correctamente su diseño para que su implementación sea lo más rápida y eficiente posible. En este apartado se explican las fases de análisis, diseño e implementación de la herramienta, donde se exponen los detalles de cada fase y los motivos por los que se ha tomado cada decisión.

5.1. Análisis

A partir de las especificaciones detalladas en Sección 3.2, se puede empezar el análisis de la herramienta. Este análisis se divide en distintas secciones, donde se plantean distintos aspectos a tener en cuenta de cara al diseño.

5.1.1. Entidades

El sistema tiene una serie de entidades, todas pertenecientes al algoritmo evolutivo, los requisitos de las entidades son muy generales, pero se detallarán más adelante.

- Gen: Debe contener uno o más valores que pueden ser de cualquier tipo.
- Individuo: Debe contener uno o varios genes. Debe de poder evaluarse su rendimiento. Debe de contener un valor numérico que represente su *fitness*, además de ser comparable con otros individuos en base a su *fitness*.
- Población: Puede contener uno o más individuos.
- Métodos: Solo puede haber una instancia de cada tipo de método (selección, reproducción y mutación durante la ejecución de la herramienta.

5.1.2. Interfaces

Para hacer que la herramienta sea lo más modular posible, es una buena idea hacer uso de interfaces, de este modo se puede abstraer la base que comparten las entidades de un mismo tipo para que el núcleo de la herramienta no se preocupe de qué tipos está manejando.

Las interfaces corresponden a cada tipo de entidad mencionada arriba (con excepción de la población, si bien esta podría también ser una interfaz), por tanto se contaría con cinco interfaces: Gen, Individuo, Selección, Reproducción y Mutación. En el caso de las tres últimas, estas podrían ser sustituidas por clases abstractas.

5.1.3. Plantillas y herencia

Uno de los requisitos esenciales de la herramienta es que sea capaz de ser usada con cualquier videojuego. Para esto es necesario que los individuos y los genes sean capaces de trabajar con distintos métodos de representar la información contenida.

Esto se puede hacer usando plantillas o herencia, para esto la clase Gen tiene que manejar la información como si fuera un objeto genérico y abstraer el procesamiento de toda esta información de modo que se encuentre dentro de las subclases, haciendo que los individuos que contienen a esos genes no necesiten trabajar directamente con ellos.

En el caso de los métodos de reproducción y mutación, es necesario manipular los genes a nivel interno, modificando los valores que almacenan. Para hacer esto lo más abstracto posible existen dos enfoques muy diferentes. Por un lado se puede delegar este cambio dentro del propio gen, esto permite que los métodos sean totalmente genéricos y no requieran conocer del tipo de gen. Por otro lado, se pueden incluir estos cambios dentro de los propios métodos de reproducción y mutación, lo que hace que estos requieran conocer el tipo de gen que manejan, pero a cambio también permite que se pueda manipular los genes de una forma más personalizada.

Para esta herramienta se usará la segunda opción, ya que esto permite crear de distintos tipos de método para cada tipo de individuo. El objetivo es crear dos ó tres métodos que modifiquen los individuos de forma distinta para compararlos.

5.1.4. Población

La población requiere almacenar los distintos individuos de cada generación. Debe de ocuparse de gestionarlos y evaluarlos, así como adaptar su *fitness*. Además, también se puede usar para pasar los individuos a los métodos.

5.1.5. Puntuación

Para que la herramienta funcione, es necesario poder asignarle una función de *fitness* para que evalúe a los individuos, esto implica que hay que obtener valores que representen distintos estados/puntuaciones del juego para dicha función.

Para esto se pueden crear unos observadores que se ocupen de detectar el estado del juego y devolverlo cuando sea necesario. Estos observadores deberán estar ligados al individuo correspondiente y detectar el estado de la instancia del juego de ese individuo.

5.1.6. Factoría de métodos

Dado que se disponen de múltiples métodos de selección, reproducción y mutación, usar un patrón de factoría para crear los distintos métodos es un modo de facilitar su gestión. También puede ayudarse de la interfaz gráfica para seleccionar el tipo de método y crearlo a partir de la factoría.

5.1.7. Adaptador de individuo

Probablemente los videojuegos que usen esta herramienta sean muy distintos entre si, esto implica que la información que necesitan representar los individuos también lo sea. Por ello, es necesario crear un adaptador que permita pasar la información del videojuego al algoritmo. Este adaptador se ocupa de llevar la representación de la IA del videojuego al individuo que usará el algoritmo evolutivo, convirtiendo esa información en algo que pueda procesar el individuo. Esto puede ser muy difícil de generalizar, ya que cada juego puede ser completamente distinto.

5.1.8. Algoritmo evolutivo

La pieza central de la herramienta es el algoritmo evolutivo, este tiene un diseño básico, posee una población, los distintos métodos, y las variables de control del algoritmo. En cuanto a comportamiento, se ocupa de ejecutar un algoritmo evolutivo clásico: Evaluación $\rightarrow$ Selección $\rightarrow$ Reproducción $\rightarrow$ Mutación $\rightarrow$ Evaluación.

Además, es el que se ocupa de obtener los resultados finales y almacenarlos. Esto se podría hacer en una clase controlador externa pero por simplicidad, será el propio algoritmo el que se ocupe.

5.2. Diseño

Debido a la naturaleza de la herramienta, pensada para ser utilizada de manera sencilla y sin muchos conocimientos de programación, el diseño

está orientado a ser lo más abstracto y modular posible, facilitando así la incorporación de nuevos elementos a la misma.

Por ello, la herramienta está estructurada de modo que se puedan agregar nuevas clases sin modificar el comportamiento interno del algoritmo, así como poder enviar la información desde la interfaz de *Unity* hasta el algoritmo.

Dado que la herramienta es una extensión de *Unity*, el diseño está centrado en el funcionamiento interno más que en la parte visual. Además, también es innecesario plantear una arquitectura multicapa o compleja (más allá de una división del núcleo de la aplicación y los diferentes módulos que la conforman) dado que toda la interfaz y los datos está gestionada por *Unity*.

Por todo esto, este apartado se centrará en el diseño de la arquitectura en cuanto a clases y patrones utilizados, así como las razones que llevaron a tomar tales decisiones.

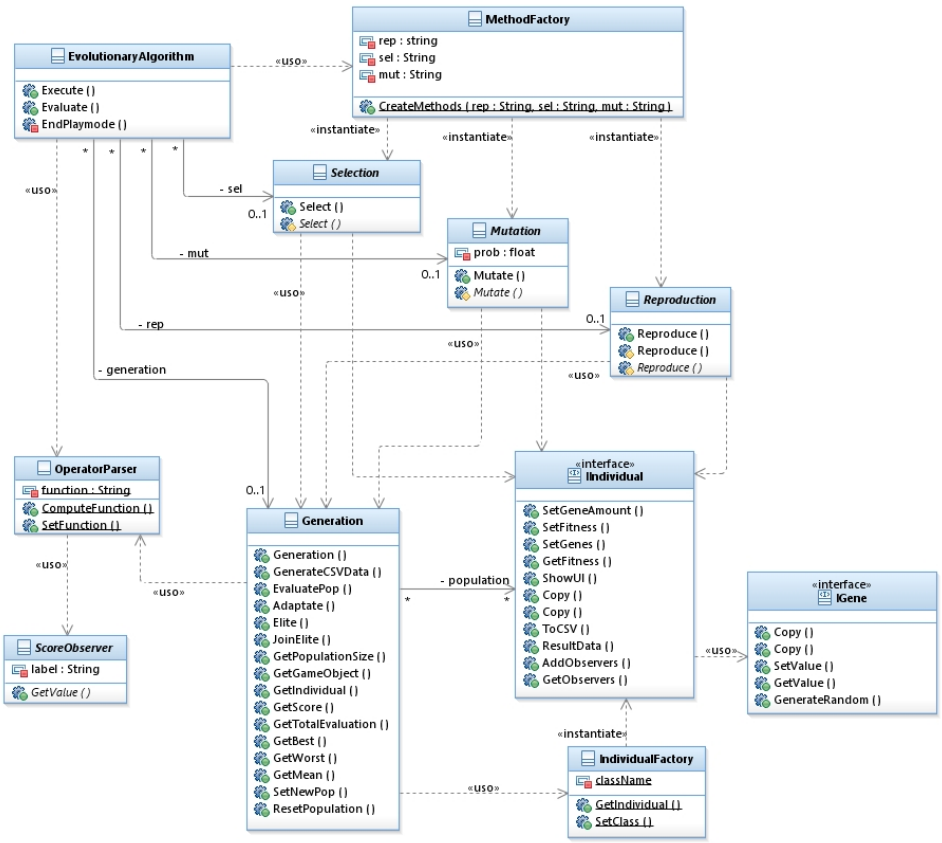

Figura 5.1: Diagrama de clases mostrando la estructura básica del algoritmo.

Como ya se ha comentado y se puede comprobar en Figura 5.1, el pilar central de la herramienta es el algoritmo evolutivo. En el diagrama se han omitido variables y funciones que no sean necesarias para entender su

funcionamiento para facilitar la lectura. El algoritmo tiene que recibir los parámetros y generar la población de individuos, ejecutar los métodos de selección, mutación y reproducción, así como pasar todos los parámetros necesarios a los otros componentes de la herramienta.

Por otro lado, también es necesario que el algoritmo pase los parámetros al resto de componentes. Esto se realiza mediantes distintos patrones que se comentarán en los respectivos componentes. Estos componentes se encuentran divididos en carpetas/paquetes, dependiendo de su tipo. Por ello tenemos *EvolutionaryComputation*, que incluye el algoritmo evolutivo, y los *sub namespaces Individuals*, *Genes*, *Mutations*, *Reproductions* y *Selections*, cada uno conteniendo las clases relacionadas. De este modo se organizan mejor las clases y se evita colisión con el nombre de otras clases que no pertenezcan a la herramienta.

5.2.1. Algoritmo evolutivo

Al ejecutarse en tiempo real o acelerado, durante la ejecución del videojuego en cuestión, requerirá el uso de las corrutinas *Awake*, *Start* y *Update* que nos proporciona *Unity* y que nos permiten realizar acciones al crear la instancia del objeto al que pertenece este *script*, al iniciar el videojuego antes que *Update* empiece su ejecución y durante toda su ejecución, respectivamente.

Esta clase dispone de los siguientes métodos para su ejecución:

- *Awake*: Se utiliza la corrutina *Awake* para instanciar los elementos visuales del objeto que mostrará el temporizador. Este temporizador se puede ajustar desde la herramienta para que el videojuego avance y se calculen los datos de los individuos durante un tiempo determinado antes de pasar a la siguiente generación ejecutando el Algoritmo Evolutivo.

- *Start*: Esta corrutina es llamada después de instanciar todos los objetos de la escena de *Unity*. Se encarga de inicializar la primera generación, el temporizador y de crear las instancias para los diferentes métodos que se podrán ejecutar a través de una cadena de caracteres.

- *Execute*: Este método ejecutará el algoritmo evolutivo tantas veces como generaciones desee calcular el usuario.

  Para su primera ejecución deberá ejecutar la corrutina *Evaluate* en tiempo real, la cuál se especificará en otro punto. Una vez hecha la primera evaluación dependerá de si necesita otra evaluación o no, en caso de necesitarla se ejecutarán los métodos de Selección, Reproducción y Mutación que modificarán la generación y luego se iniciará la corrutina *Evaluate*. En caso contrario se copiarán los datos del resultado

en formato *CSV* y se pasará a la siguiente iteración. En ambos casos dependiendo de si el usuario ha seleccionado la opción que permite el uso de Elitismo, antes de hacer la evaluación conseguirá los individuos elite de la generación y después de hacer la evaluación juntará estos individuos elite a la generación actual.

Al finalizar la ejecución del algoritmo evolutivo sobre todas las generaciones requeridas se detendrá el videojuego mediante el uso de la función *EndPlayMode* y se creará un fichero con los resultados del algoritmo en formato *CSV*.

- *CreateCSVFile*: Escribe en un fichero los datos en formato *CSV* que conseguimos durante la ejecución del algoritmo.

- *Update*: En esta corrutina que se ejecutará continuamente se tomará en cuenta si se ha marcado la opción de ocultar los elementos de la escena de *Unity* para consumir recursos, además de iniciar la ejecución del algoritmo evolutivo en caso de ser necesario.

- *Evaluate*: Se trata de otra corrutina que cambiará la escala de tiempo y permitirá realizar la evaluación de la población perteneciente a la generación actual en la escala de tiempo seleccionada. Para realizar esta evaluación llamará al método *EvaluatePop* perteneciente a la generación.

- *EndPlaymode*: Detiene la ejecución del videojuego en *Unity* y es llamado tras ejecutar el algoritmo durante el número de generaciones deseadas.

5.2.2. Genes

Para poder ofrecer nuestra herramienta al mayor número de usuarios posibles se ha creado una estructura genérica que nos permitirá extender el tipo de gen a cualquiera que requiera el usuario, para lo que solo deberá crear una nueva clase que herede de *Gene* (Figura 5.2), que a su ver implementa la interfaz *IGene*.

Esta clase *Gene* nos permitirá copiar estos genes, de manera que no ocurran problemas de memoria, generar valores aleatorios que requerirá nuestro algoritmo y poder acceder y cambiar sus valores.

La clase *Gene* es una clase abstracta que implementa la interfaz *IGene*. Consiste en una constructora que genera un nuevo gen aleatorio, métodos *getters* y *setters*, además del método *copy* que introduce una copia del valor de otro gen en el actual para evitar conflictos de memoria a la hora de acceder a valores por referencia.

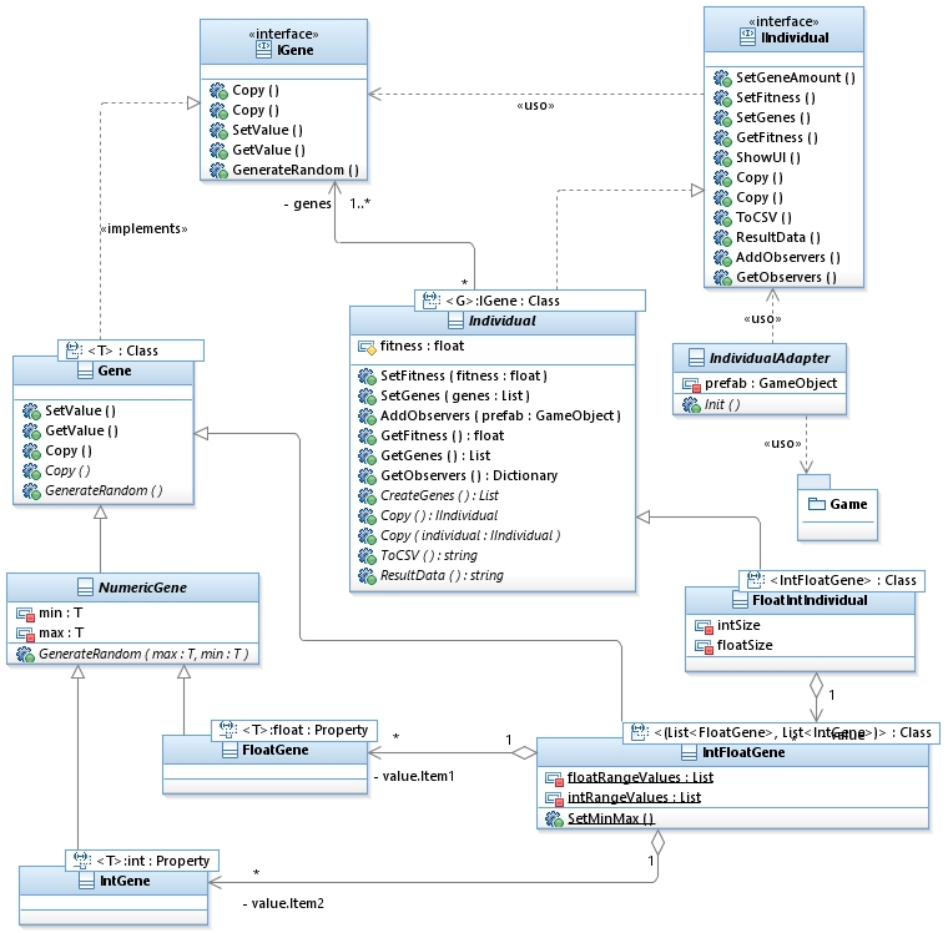

Figura 5.2: Diagrama de clases mostrando el individuo y los genes.

Como base de la herramienta se plantean los siguientes tipos de genes al ser los más comunes y sencillos de utilizar en muchos tipos de juegos diferentes:

- *NumericGene*: Clase abstracta que hereda de la clase *Gene*. Esta clase funciona como una interfaz para otras clases de genes de tipo numérico. Por tanto implementa una constructora que devolverá un gen aleatorio a partir de unos valores mínimo y máximo a partir de la clase abstracta *GenerateRandom*.

- *IntGene*: Gen de números enteros. Hereda de la clase *NumericGene* y consta de dos constructoras, una vacía la cual generará un rango de 0 a 1 y otra que recibirá una pareja de valores mínimo y máximo. También implementamos los métodos de copia para un gen de tipo *Integer* y *GenerateRandom* que generará un valor aleatorio entre estos

valores mínimo y máximo.

- *FloatGene*: Gen de números de tipo coma flotante. Al igual que *IntGene* implementa los mismos métodos, con la única diferencia siendo el tipo de valor que utiliza.

- *IntFloatGene*: Este tipo de gen es un poco diferente, ya que es muy común que los videojuegos utilicen parámetros de ambos tipos tanto *Int* como *Float*. Por esto implementamos esta clase que toma como valores dos listas de parámetros, una de tipo *Int* y otra de tipo *Float*. Tiene dos constructoras al igual que las otras clases donde se inicializan estás listas, dependiendo de si se le quiere pasar el número de genes o no. Implementa dos métodos de copia, uno que genera una copia del gen actual y otro que recibe un parámetro gen que se quiera copiar dentro de este asignando su valor. También implementa los métodos *GenerateRandom*, que rellena las listas con valores aleatorios que se encuentren dentro del rango establecido para este gen, y el método *SetMinMax* que establece los rangos anteriormente mencionados con los parámetros deseados.

5.2.3. Individuos

Disponemos de una interfaz *IIndividual* (Figura 5.2) donde establecemos todos los métodos de las que requiere un individuo para poder ser utilizado por la clase de generación para establecer una población de individuos. Entre estos métodos encontramos *getters* y *setters* que nos permitirán gestionar el número de genes, los genes en específico del individuo y su valor *fitness*. Se crean dos métodos de copia al igual que los genes, uno para copiar el valor del individuo actual y otro para copiar otro individuo dentro del actual. Individuo necesitará algunos métodos de los que no disponíamos en los genes como:

- *ShowUI*. Nos permite guardar un booleano en caso de que el usuario no desee que se vea la interfaz de la instancia del videojuego para consumir menos recursos del sistema al ejecutar nuestro algoritmo evolutivo.

- *ToCSV*. Este método nos devolverá los datos del individuo cogiéndolos del observador, que los mantendrá actualizados, en formato *CSV* con sus valores separados por comas.

- *ResultData*. Utilizamos este método para conseguir los resultados de cada gen del individuo en un formato de texto que podremos mostrar al usuario de manera entendible.

- *AddObservers*. Se encarga de añadir los elementos que contienen las puntuaciones del videojuego al diccionario de observadores de la clase

de individuo. De esta manera dispondremos de las puntuaciones actualizadas a la vez que avanza el videojuego.

- *GetObservers*. Nos devuelve el diccionario de los observadores de las puntuaciones anteriormente mencionados.

Ya con nuestra clase interfaz definida podemos hablar de los diferentes tipos de individuos que hemos implementado como base por ser comúnmente utilizados y que a la vez utilizaremos en nuestro ejemplo de ejecución de la herramienta:

- *Individual*: Es una clase abstracta que se encarga de implementar los métodos más genéricos de la interfaz de individuos. En esta clase tenemos una constructora que se encarga de generar los genes del individuo, además de inicializar su lista de observadores de puntuaciones. También implementamos los *getters* y *setters* antes mencionados en la interfaz y las funciones relacionadas con los observadores de las puntuaciones.

- *FloatIntIndividual*: Esta clase hereda de la clase abstracta Individual e implementa las funciones restantes, más específicas de su tipo, que será tanto *Float* como *Int*. Es necesario implementar los dos tipos de valores debido a que por un lado las puntuaciones usualmente las manejamos como números enteros, pero para los valores de los parámetros de los elementos del juego tendemos a usar valores de tipo *Float* que nos dan una mayor precisión.

  Implementamos los métodos de copia, tanto de copiar los datos del individuo, como copiar dentro de este individuo datos de otro, el método para crear genes del individuo, el cuál crea una lista de genes del tipo necesario y reserva espacio para esta, además de los métodos antes comentados *ToCSV* y *ResultData*, que nos devuelven datos de los resultados del individuo en formato de texto, los cuales se podrán exportar y contrastar con los resultados requeridos por el usuario.

5.2.4. Mutaciones

La clase *Mutation*, mostrada en Figura 5.3, está compuesta por una constructora que recibe la propiedad de probabilidad para que ocurra esta mutación, una función *Mutate* que a partir de una generación de individuos aplica esta mutación a cada uno de ellos en base a la probabilidad especificada, y una función abstracta llamada *Mutate* de igual manera pero que recibe un individuo como parámetro, esta función se implementará dentro de las clases que hereden de *Mutation* y apliquen una mutación específica.

Para utilizar en nuestros ejemplos implementaremos tres clases diferentes de mutaciones, pero siempre que el usuario de la herramienta lo necesite, podrá añadir los tipos de mutaciones que le parezcan convenientes.

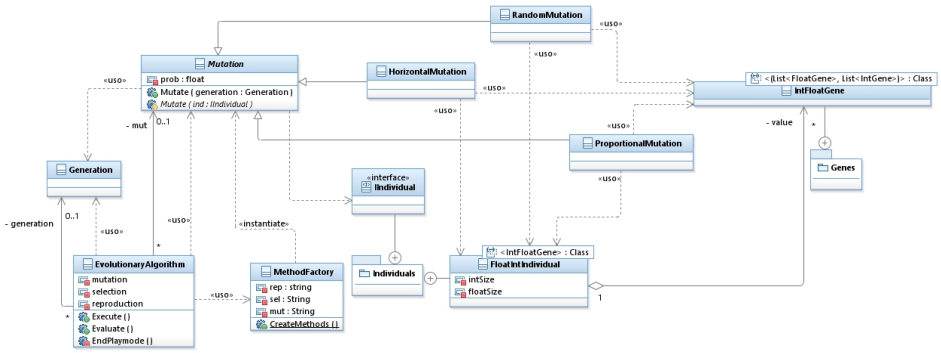

Figura 5.3: Diagrama de clases mostrando la mutación.

Estas clases heredan de *Mutate* e implementan su método *Mutate* recibiendo un individuo como parámetro, este individuo será de tipo *FloatIntIndividual* y ejecuta el algoritmo de mutación que se aplicará a todos los genes del individuo, tanto genes de tipo *Int* como genes de tipo *Float*.

- *RandomMutation*: Se utiliza una mutación completamente aleatoria por defecto. Esta mutación tendrá un veinte por ciento de probabilidad de cambiar el valor de un gen por otro valor aleatorio dentro de un rango establecido.
- *HorizontalMutation*: La mutación horizontal como nosotros nos referimos a ella va a disminuir el valor de un gen o aumentarlo por dos de manera aleatoria y con una probabilidad del 50 por ciento.
- *ProportionalMutation*: En la mutación proporcional los valores aumentarán o disminuirán en una pequeña proporción de cuatro u ocho veces su valor actual con un 40 por ciento de probabilidad de aumentar y el resto de disminuir.

5.2.5. Reproducción

La clase abstracta *Reproduction* (Figura 5.4) servirá de base para los diferentes métodos de reproducción que iremos necesitando o consideramos útiles y que nos pueden devolver buenos resultados.

La clase *Reproduction* no cuenta con una constructora, sino que se implementan diferentes clases *Reproduce* que recibirán diferentes tipos de parámetros:

- *Generation generation*: Recibe una generación por parámetro y se encarga de inicializar una lista con los padres de un individuo, además llamará a la función que recibe los padres como parámetro para conseguir la descendencia de esta generación.

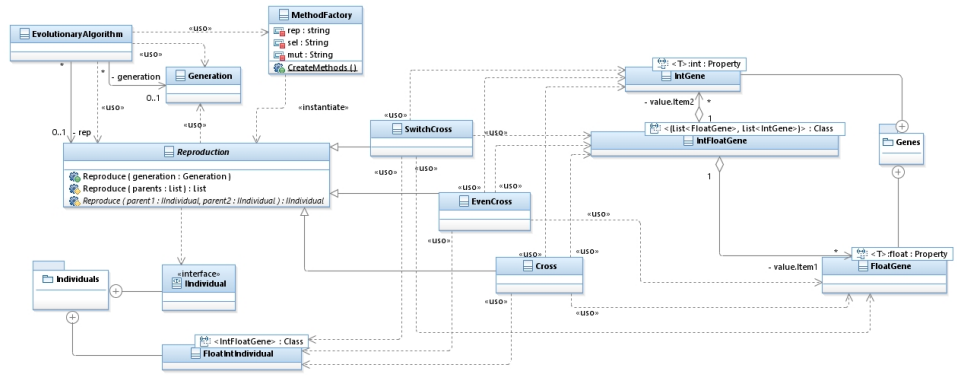

Figura 5.4: Diagrama de clases mostrando la reproducción.

- *List<IIndividual>parents*: Este método recibe una lista de padres y devolverá la descendencia de todos estos padres, para esto cogerá pares de padres con los que llamará al siguiente método el cuál devolverá al individuo hijo.

- *IIndividual parent1, IIndividual parent2*: Método abstracto que se implementará en todas las clases que hereden de *Reproduction* y se encargará de realizar la mezcla de los genes de los individuos padres para generar a un nuevo individuo hijo.

Como base para realizar las pruebas del proyecto hemos añadido las siguientes clases que heredarán de *Reproduction* e implementarán el método de reproducción de padres:

- *ColorCross*: Implementará en el método reproducción un algoritmo que se encarga de calcular la diferencia entre cada par de genes de ambos padres, la cuál se restará una cuarta parte al valor del gen perteneciente al nuevo individuo, o se sumará dependiendo si la diferencia es positiva o negativa, para los genes de tipo *Float*. Para los genes de tipo *Int* intercambiará los valores de los genes con una probabilidad del cincuenta por ciento. Este método al igual que los siguientes devolverá el primer individuo padre después de alterar sus genes en base a ambos padres. El nombre de este método era referencia a la mezcla de colores, obteniendo así valores intermedios.

- *EvenCross*: Este método calcula el valor medio de los genes de ambos padres en caso de que el índice del gen actual sea par, en caso contrario restará ambos valores y multiplicará por dos su valor absoluto. Ejecutará el mismo algoritmo tanto para los genes de tipo *Float* como para los de tipo *Int*.

- *SwitchCross*: En el método de *switch* o intercambio, en caso de que el valor del gen del primer padre sea menor que el del segundo tendrá un sesenta por ciento de probabilidades de intercambiar su valor por el del segundo padre. Igual que en el caso anterior se ejecutará el mismo algoritmo para los genes de ambos tipos.

5.2.6. Selección

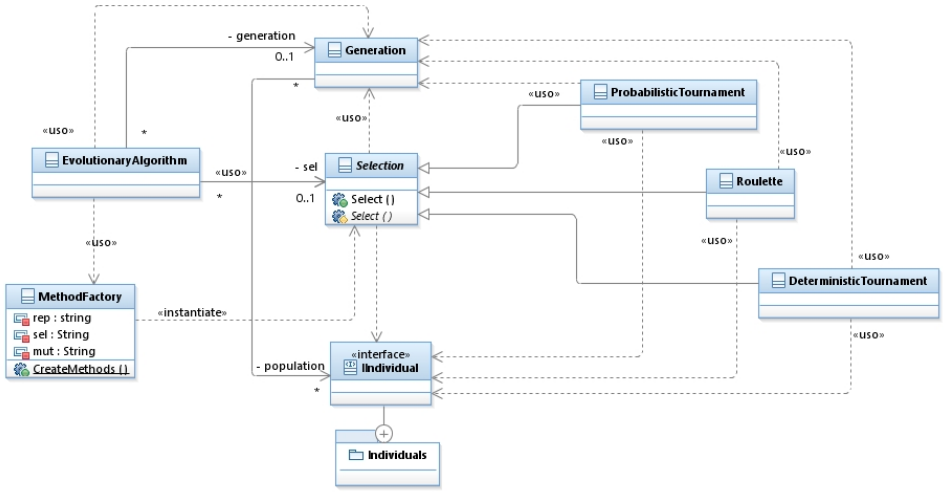

Figura 5.5: Diagrama de clases mostrando la selección.

Por otro lado debemos implementar seguros en nuestros algoritmos que prevengan que los individuos más adecuados acaparen toda la población en unas pocas generaciones, ya que esto nos llevará a tener diferentes soluciones muy similares y poco representativas, además de una gran pérdida de diversidad en nuestra población.

Para que el algoritmo evolutivo pueda darnos buenos resultados es imperativo que mantengamos una población diversa, ya que tener una población compuesta de puros individuos aptos nos llevará a una solución prematura y poco fiable.

La clase abstracta llamada *Selection* (Figura 5.5) la cuál dispone de dos métodos *Select*, el método principal y más genérico recibe por parámetro una generación y el otro método recibirá por parámetro además de la generación una colección con las probabilidades de selección de los miembros de la población perteneciente a esta generación. Este último método será abstracto y por tanto deberá ser implementado por las clases que heredarán de *Selection*.

En el caso del primer método para todos los miembros de la población de la generación que recibe, conseguirá la probabilidad de que sea seleccionado a partir de el valor del mismo entre el valor total de evaluación de la generación.

Luego sumará la probabilidad de selección a una variable donde guardaremos el total de las probabilidades de los distintos miembros de la población. Y por último guardará este total actual en una colección con las probabilidades acumuladas de los diferentes miembros de la población.

Este método inicial hará una llamada al segundo método pasándole por parámetros tanto la generación como la colección de probabilidades acumuladas, el cuál será implementado en las siguientes clases:

- *Roulette*: Selecciona la nueva población en base al giro de una ruleta, la cuál se dividirá en base a la colección de probabilidades que recibe por parámetro. Traducido a la implementación, para cada miembro de la nueva población cogerá de manera aleatoria un miembro de la población antigua. Este algoritmo de selección no suele ser recomendable, ya que si un miembro de la población tiene una probabilidad muy alta de ser seleccionado es muy posible que la nueva población este compuesta en su mayoría por el mismo, además no es posible utilizarlo con valores de *fitness* negativos. Ha sido implementado de igual manera ya que es muy sencillo de entender y nos proporcionará resultados distintos para comparar con otros algoritmos mejores.

- *Tournament*: El algoritmo de torneo tomará un rango de individuos dentro de una población por cada miembro de esta población, en nuestro caso el rango será de tres miembros de la población, ya que este es el número mínimo para poder realizar un torneo; luego a partir de este rango conseguirá el mejor individuo en base al tipo de torneo, el cuál introducirá en la nueva población como un padre.

  Este algoritmo es muy popular cuando se trata de programación evolutiva ya que permite utilizar valores de *fitness* negativos. Además de devolver muy buenos resultados en general.

  Para la herramienta se implementarán dos tipos de algoritmos de torneo:

  - *DeterministicTournament*: El torneo determinista seleccionará el individuo con el valor más alto del rango y lo introducirá en la nueva población como un padre. Este proceso se repetirá para el número de individuos de la población anterior.
  - *ProbabilisticTournament*: En el torneo probabilístico añadimos cierta aleatoriedad que nos permitirá conseguir mayor diversidad en la nueva población. De manera que seleccionamos en lugar del valor más alto, bien el valor máximo o el valor mínimo del rango de individuos con una probabilidad del cincuenta por ciento. Al igual que en el torneo determinista, se añadirán estos individuos como padres a la nueva población.

5.2.7. Generación

La generación encapsula una población de individuos y contiene todos sus datos relevantes. El algoritmo evolutivo se realizará durante muchas generaciones buscando mejorar el resultado de esta en cada iteración.

Dentro de la generación podremos calcular los valores globales de la población, como por ejemplo, calcular cuales serán los mejores o peores individuos, sus valores de *fitness* y los valores totales. Incluso permitirá exportar los resultados de esta población en formato *CSV* para poder clasificarlos y generar gráficas representativas.

La generación usa la interfaz *IIndividual* para gestionar los individuos, abstrayendo así el comportamiento interno de los mismos, de este modo una sola clase *Generation* puede gestionar cualquier implementación de *IIndividual* por compleja que sea.

La clase *Generation* necesita los siguientes métodos:

- *Generation*: Constructora que se encargará de inicializar todas las variables globales que necesitaremos en esta clase.
- *GenerateCSVData*: Método que toma los datos necesarios para representar los resultados de la generación y los concatena en formato *CSV*.
- *EvaluatePop*: Método que evalúa la población, de manera que reinicia los valores y los recalcula comprobando los datos de todos los individuos de la población que pertenece a esta generación. Conseguirá los mejores y peores individuos, además del cálculo global de la población. Una vez ha calculado los nuevos datos hace una llamada al siguiente método *Adaptate*.
- *Adaptate*: Este método es utilizado al evaluar la población y se encarga de adaptar los valores de resultado en base al *fitness* del peor individuo.
- Elite: Consigue la élite de una población, esta élite será un porcentaje de la población, el cuál se pasará como un parámetro ya que podrá ser modificado por el usuario; estará compuesta por los mejores individuos de la población en el porcentaje determinado.

  Al utilizar elitismo nos aseguramos de que, aunque el algoritmo tenga aleatoriedad, se mantengan los mejores resultados entre generaciones. El elitismo puede ser contraproducente dependiendo del resultado que se necesite por lo que su uso es opcional en esta herramienta.
- *JoinElite*: Método que introduce a los valores élite dentro de la población actual descartando a los peores individuos de esta población.
- *ResetPopulation*: Restablece los valores de todas las variables que pertenecen a la clase *Generation*.

Además dispondrá de diferentes *getters* y *setters* para muchos de estos elementos, ya que serán modificados por el algoritmo evolutivo.

5.2.8. Puntuación

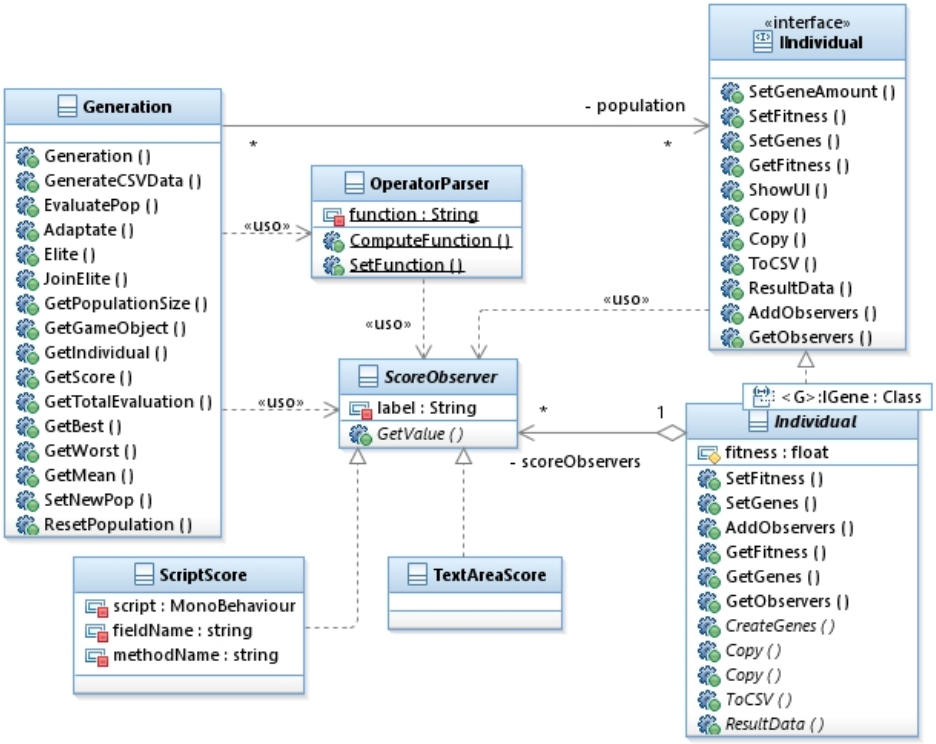

Figura 5.6: Diagrama de clases mostrando el *OperatorParser* y el *ScoreObserver*.

Para que la herramienta *EvoUnity* funcione, tal y como se plantea en Subsección 5.1.5, es necesario que sea capaz de encontrar los valores que representan las variables de la función de *fitness*. Para esto se opta por un patrón observador compuesto de la clase abstracta *ScoreObserver* (Figura D.8), que dispone de una variable *label*, que indica el nombre de la variable que representa en la función de *fitness*, y un método *GetValue* que se ocupa de obtener el valor.

La forma de obtener el valor de cada *ScoreObserver* depende de cómo se quiera implementar. En este caso se plantean dos clases: *ScriptScore* y *TextAreaScore*, que obtienen los valores a partir de otro *script* del proyecto o de un texto de la interfaz de usuario.

Además, también hace falta un *OperatorParser*, que se ocupará de gestionar la función de *fitness*, recibiéndola mediante una cadena de caracteres

y convirtiéndola en una función matemática que se pueda ejecutar.

OperatorParser dispone de dos métodos estáticos:

- *SetFunction*: Asigna la función al *OperatorParser*.
- *ComputeFunction*: Recibe los *ScoreObserver* de un individuo y calcula su *fitness* en base a la función que tenga almacenada.

Como ya se ha mencionado, los *ScoreObserver* son almacenados en el individuo por la clase *Generation*.

5.2.9. Factoría de métodos

Para instanciar los distintos métodos de selección, reproducción y mutación es necesario una factoría que permita generar las diferentes clases en tiempo de ejecución. La clase *MethodFactory* se ocupa de crear las instancias de mediante la función *CreateMethods*, que los generará a partir del nombre de la clase representado en una cadena de caracteres.

5.2.10. Temporizador

El la herramienta *EvoUnity* requiere de un temporizador, *Timer*, que se ocupe de calcular el tiempo restante de la ejecución del algoritmo y mostrarla por pantalla. Para esto, *Timer* recibe el tiempo de ejecución y lo reduce hasta llegar a 0. A su vez, debe de ser capaz de mostrar ese tiempo en la interfaz de usuario.

5.3. Implementación

En esta sección se van a detallar las peculiaridades sobre la implementación de la herramienta *EvoUnity* diseñada para *Unity* en su versión 2020.1.9f1, utilizando el lenguaje de programación C#.

Para que la *EvoUnity* pueda ser utilizado por una mayor cantidad de usuarios, sus clases y métodos deben ser genéricos, dando la posibilidad de utilizar cualquier tipo de valores dependiendo de las necesidades del usuario. Por tanto se crearon las diferentes clases explicadas en el apartado de diseño 5.2 usando tipos genéricos para sus parámetros.

- Genes. Los genes se implementan utilizando el tipo genérico <T>, este tipo genérico representa el valor que contiene el gen. De este modo los genes pueden contener cualquier tipo de valor que se pueda contener en una variable.

- Individuos. Los genes son los valores que maneja cada individuo, por tanto para su implementación se asignó el tipo genérico <G>que hereda de gen. Cada individuo guardará una lista de genes de G para poder realizar sus operaciones. Todas estas operaciones se realizan a partir de esta clase plantilla Además se implemento algunas clases auxiliares necesarias para el manejo de estos individuos desde tanto el editor de *Unity* que será la parte visual de nuestra herramienta, como desde la clase que se encarga de ejecutar el algoritmo *Generation*.
  - *IndividualFactory*. Clase que utiliza el patrón factoría y permite generar instancias de individuos.
  - *IndividualHandler*. Esta clase ayuda a inicializar los individuos a través de *IndividualFactory* desde el editor de *Unity*.
- Mutaciones y reproducciones. Al implementar las subclases de mutación y reproducción se debe especificar el tipo de objeto que se utilizará en el videojuego concreto para el correcto funcionamiento del algoritmo. Las subclases implementadas para las pruebas se realizaron en base a genes de tipo *Int* y *Float*, por lo que para cada gen de estos tipos se realiza una mutación o reproducción concreta.
- Selección. Las subclases de selección fueron implementadas siguiendo los algoritmos de los métodos más utilizados en la computación evolutiva y adaptando su uso a C#.
- Generación. Al realizar la implementación de la generación se utiliza el tipo genérico Individuo para mantener una población, además de los atributos de esta.
- Puntuación. Para mantener tanto los valores de todos los individuos de la población como los valores totales actualizados, se implementó un observador que notificará los cambios realizados sobre estos valores. Este observador, *ScoreObserver* será una clase abstracta que contendrá un único método abstracto *GetValue*, este método permitirá conseguir el valor actual del individuo. Se utiliza una clase abstracta en lugar de una interfaz para facilitar la depuración y poder observar estos valores en tiempo real mediante la consola de *Unity*. Las subclases que heredan del observador de puntuación e implementarán el método abstracto *GetValue* serán:
  - *ScriptScore*: Clase genérica que permitirá coger el valor de un *script* seleccionando un nombre de campo o de método que devolverá el valor.
  - *TextAreaScore*: Esta clase permitirá coger el valor directamente de un área de texto que se encuentre añadida a la interfaz del videojuego en cuestión.

Este observador será muy útil para manejar estos valores desde el editor de *Unity* a través de la clase *ScriptScoreEditor* en la cuál se implementará una de las partes visuales de la herramienta *EvoUnity* a la cuál tendrán acceso los usuarios.

- Factoría de métodos. Clase que utiliza el patrón Factoría para encapsular los diferentes tipos de métodos utilizados por el algoritmo evolutivo. Permitirá crear o conseguir instancias de una tupla de tres elementos, la cuál contendrá los tres tipos de métodos necesarios a la hora de ejecutar el algoritmo evolutivo: Mutación, Selección y Reproducción. Recibirá como parámetros *strings* que contendrán los diferentes métodos de cada tipo, además de una matriz de opciones en caso de ser necesarias para crear estos métodos, como por ejemplo, la probabilidad de que ocurra un fenómeno que lleve a un cambio dentro del método. Esta clase cuenta con los métodos principales que permiten crear o devolver las instancias de los diferentes métodos:

  - *CreateMethods*: Recibe por parámetros los tres tipos de métodos que se necesitan al igual que las opciones. Descompone los parámetros por sus tipos para encontrar sus subclases y a partir de estas crea las instancias por cada tipo de subclase. Devolverá una tupla de tres elementos, cada una con las diferentes instancias para cada subclase.
  - *GetMethods*: Este método recibe por parámetros *strings* para cada tipo de método que se quiere conseguir. Descompondrá estos *strings* para conseguir los diferentes tipos de métodos y a partir de estos tipos ya nos será posible conseguir las diferentes instancias.

Cabe destacar que para varias de estas clases anteriormente mencionadas se implementaron métodos de copia. Para realizar estas copias se crean objetos nuevos en lugar de utilizar el mismo objeto original, de esta manera se evita la modificación de datos en objetos equivocados.

Además de las clases anteriores descritas en el apartado de diseño 5.2, también se implementaron las clases necesarias para representar el apartado visual de la herramienta *EvoUnity* utilizando editores personalizados.

5.3.1. Editores Personalizados o *Custom Editors*

Los editores personalizados[1] son clases necesarias para gestionar el apartado visual de la herramienta, las cuales se encontrarán en el paquete Editor del proyecto. Estas clases añaden una capa de funcionalidad por encima de los componentes necesarios a la hora de ejecutar el algoritmo evolutivo:

[1] `https://docs.unity3d.com/Manual/editor-CustomEditors.html`

- *MethodEditor*: Este editor se encarga de mostrar los selectores de la herramienta en el editor de *Unity* por encima de la funcionalidad de *EvolutionaryAlgorithm*. Estos selectores incluirán los diferentes tipos de selección, reproducción y mutación implementados. Además dispondrá de un botón que actualizará esta lista en caso de haber añadido o eliminado algún elemento, ya que el usuario podrá añadir nuevos tipos de selección, reproducción y mutación dependiendo de sus necesidades y sus conocimientos sobre el tema. El algoritmo evolutivo utilizará los tipos de selección, reproducción y mutación seleccionados en el editor durante su ejecución.
- *IndividualHandlerEditor*: Editor para la clase *IndividualHandler* que permite controlar a los individuos de una población desde la herramienta. Cuenta con un selector que contiene todos los tipos de individuos implementados, una vez seleccionado el tipo de individuo permitirá añadir el número de individuos necesarios y se mostrará una lista para incluir los valores mínimos y máximos iniciales para cada individuo.
- *ScriptScoreEditor*: Editor que expande la funcionalidad de *ScriptScore* para conseguir el valor de un elemento. Ha sido implementado de manera genérica por lo que permitirá conseguir un valor dentro de un *script* que pertenezca al mismo objeto que contiene *ScriptScore*. A partir de este *script* se podrá seleccionar si el valor se debe conseguir directamente de un campo perteneciente al *script* o si se consigue a través de un método, luego se podrá seleccionar si utiliza un *getter* para conseguir el valor o un contador para llevar la cuenta del valor y por último nos permite seleccionar el método en específico que devolverá el valor.

5.4. Pruebas

En esta sección se van a explicar las modificaciones necesarias sobre los videojuegos seleccionados, *Warrior Defense* y *MicroRTS*, para su correcto funcionamiento y la implementación de la herramienta sobre los mismos.

5.4.1. *Warrior Defense*

En un principio se optó por el videojuego *Warrior Defense* por su buena presentación y su uso de factorías para la generación de las unidades del juego, pero poco después de la realización de una demo de la herramienta y su aplicación sobre este juego surgieron varias carencias y fallos que estaba presentando.

- Sistema de puntuaciones: *Warrior Defense* no contaba con un sistema de puntuaciones, por lo que en base a la actuación de la inteligencia

artificial utilizada en este juego, se implemento un sistema de puntuación que sumará la vida restante de cada invocación que logre cruzar el territorio enemigo.

- Sistema de energía: Ya que los personajes del videojuego invocaban luchadores en un tiempo específico y sin ninguna estrategia, se añadió un sistema de energía que se recarga con el tiempo de manera constante y que permite realizar las invocaciones si se dispone de la energía necesaria para esta.

- Cambios en el mapa de juego: Se realizaron cambios en el mapa para que se pudiera representar el modo de juego deseado, de esta manera creando los territorios contrarios.

- Interfaz de usuario: Se añadió una interfaz de usuario para mostrar las puntuaciones de los adversarios, barras de energía para ambos que les permitirán realizar las invocaciones, y por último un temporizador que mostrará la duración de la partida hasta que se genere una nueva generación con los resultados de la anterior y empiece de nuevo.

  En cierta parte de la implementación se añadieron también elementos que representaban las invocaciones que podía realizar el usuario, pero debido a que el número de estas invocaciones aumentó para realizar pruebas con mayor diversidad, además de que durante el uso de la herramienta se buscará ejecutar un gran número de instancias del videojuego por lo que la interfaz de usuario pasa a ser algo secundario.

- Inteligencia artificial aplicada a los invocadores: En el videojuego existía una inteligencia artificial de base que permitía a las invocaciones interactuar entre ellas para encontrarse y empezar una pelea, pero los invocadores tenían una funcionalidad muy básica que solo permitía realizar una invocación cada cierto tiempo de un luchador aleatorio.

  Para solucionar esto se implemento una inteligencia artificial que permite a los invocadores decidir que tipo de luchador invocar en base a la cantidad de energía disponible, además de tomar en cuenta el último luchador invocado por el contrincante para dar prioridad a un tipo de luchador que tenga mayor ventaja.

5.4.2. *MicroRTS*

El segundo videojuego elegido fue *MicroRTS* por las siguientes razones:

- Pertenece a un género de juego completamente distinto y mucho más complejo que el *Warrior Defense*, siendo este un *Real time strategy* (RTS).

- Un diseño casi completo con facilidades para crear una inteligencia artificial sencilla.
- Facilidad de acceso y libertad de uso al ser un proyecto proporcionado por nuestro director de proyecto.

A pesar de todo eso hubo que realizar diversos cambios en el juego *MicroRTS* para que la herramienta funcionara correctamente:

- Creación de *prefabs*[2] que engloben la escena del juego.
- Instalar un paquete nuevo al proyecto que permita añadir un *NavMesh*, terreno que define donde pueden navegar objetos con IA implementada, a un *prefab* a través del componente *NavMesh Surface*[3].
- Modificar todos los objetos que utilicen el patrón *Singleton*[4] y dependan de estos para funcionar como parte del individuo que se instanciaría múltiples veces.
- Creación de una IA simple que explote en gran medida la capacidad de generar combinaciones de la herramienta. Para ello esta IA se divide en 3 partes:
  - Generar unidades continuamente, hasta llegar al máximo permitido por sus genes.
  - Una lista de movimientos disponibles a realizar con cada una de las unidades posibles, estos movimientos serán decididos por el algoritmo evolutivo a lo largo de las iteraciones. Existe un total de 9 posibilidades.
  - Una lista de todos los destinos posibles respecto a esa unidad elegida, estos también serían elegidos por el algoritmo evolutivo. Existe un total de 28 posibilidades por unidad.

  Teniendo en cuenta la cantidad de posibilidades en cada movimiento realizado, el número de combinaciones posible asciende a niveles muy altos.

En este juego a pesar de no existir un sistema de puntuaciones se decidió utilizar un *script* propio de la herramienta para obtener la puntuación. Permitiendo mostrar como obtener los datos sin necesidad de un sistema de puntuación implementado dentro del propio juego.

[2] `https://docs.unity3d.com/Manual/Prefabs.html`
[3] `https://docs.unity3d.com/Manual/class-NavMeshSurface.html`
[4] `https://en.wikipedia.org/wiki/Singleton_pattern`

Capítulo 6

Resultados

En este capítulo se muestran los resultados obtenidos de las pruebas automáticas realizadas en Warrior Defense y MicroRTS, y se realiza una discusión sobre estos resultados.

6.1. Resultados

A diferencia de lo que se podría esperar con ciertos métodos de selección como *roulette*, como que las gráficas llegasen a estancarse en cierto punto, en los resultados esto no se llega apreciar en ningún caso. Probablemente es debido a la aleatoriedad propia de los juegos, donde a pesar de tener las mismas condiciones no se logran los mismos resultados siempre. A pesar de esta desventaja para obtener resultados fiables, se puede comprobar que se obtienen mejoras y resultados superiores a lo largo de la ejecución.

A continuación se mostrarán todas las gráficas obtenidas de las pruebas en los diferentes juegos de prueba. Esta es la leyenda de las gráficas.

- El eje Y representa el *fitness* o puntación, es decir, lo que se han acercado a la puntuación objetivo.
- El eje X representa el tiempo de ejecución de la herramienta. Estando representado por el numero de ejecuciones del juego objetivo.
- La linea azul representa el mejor individuo de la generación actual.
- La linea naranja representa la puntuación media entre todos los individuos.
- La linea gris representa el peor individuo de la generación.

Para realizar las pruebas se ha ejecutado la herramienta con ciertos parámetros durante un tiempo especificado. La mayoría parámetros son los mismos en cada ejecución del mismo juego, a excepción de los métodos de

mutación, reproducción y selección, al ser estos los que más varían los resultados obtenidos. Cada una de las pruebas representa una combinación de cada uno de estos métodos en cada juego.

6.1.1. *Warrior Defense*

Comenzando con el juego *Warrior Defense*. Para evaluar el rendimiento de los jugadores añadimos una puntuación basada en el número de criaturas que alcanzan el otro extremo del mapa. Las pruebas realizadas tienen como parametros en común:

- Tiempo de juego: 20 segundos.
- Número de generaciones: 20 generaciones o iteraciones del algoritmo.
- Velocidad del juego: 5 veces más rápido.
- Tamaño de la población: 20 individuos.
- Porcentaje de elitismo: 10 %.
- Probabilidad de mutación: 20 %.
- Probabilidad de reproducción: 60 %.

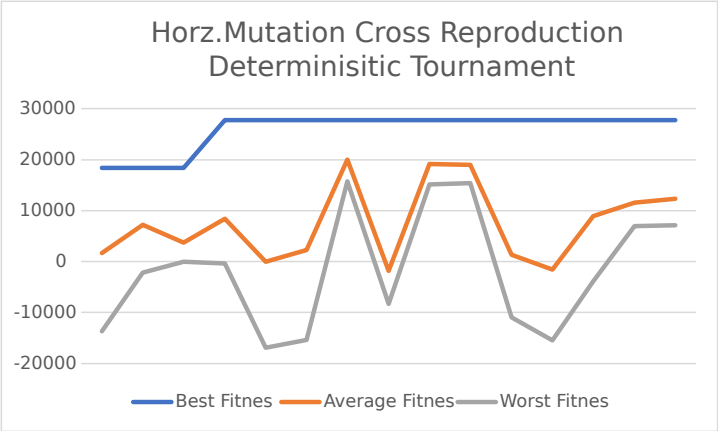

Figura 6.1: Resultados en WarriorDefense | Horizontal Mutation | ColorCross Reproduction | Determinisitic Tournament.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.1 con los parámetros:

- Método de mutación: *Horizontal*.
- Método de reproducción: *ColorCross*.

- Método de selección: *Deterministic tournament*.

En esta gráfica se puede observar una mejoría inicial hasta llegar a un máximo (Con posibilidad de mejora si se aumenta el tiempo de ejecución). La variación tan extrema que se observa a veces en la media y el peor individuo de la generación son provocados por un peor o mejor rendimiento del mismo individuo en diferentes ejecuciones, sumado a la generación de nuevos individuos diferentes por los métodos de cruce y mutación dados.

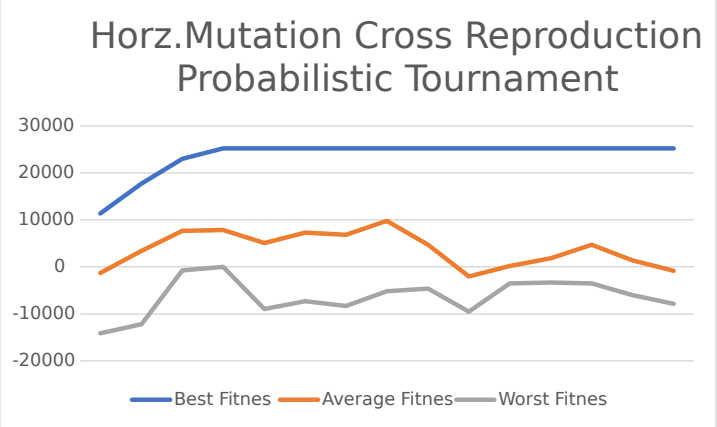

Figura 6.2: Resultados en WarriorDefense | Horizontal Mutation | ColorCross Reproduction | Probabilistic Tournament.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.2 con los parámetros:

- Método de mutación: *Horizontal*.
- Método de reproducción: *ColorCross*.
- Método de selección: *Probabilistic tournament*.

En esta gráfica podemos observar una mejoría inicial más suave con respecto a la gráfica 6.1 . En cambio esta sufre de una media y un peor individuo inferiores en puntuación a la de la gráfica 6.1. Esto es provocado por el método de selección que favorece de vez en cuando a los peores individuos. Aunque suene como algo negativo el favorecer a los peores individuos, de hecho es algo beneficioso en muchos casos a la larga. Al permitir que peores individuos continúen existiendo se amplían la cantidad de combinaciones posibles de genes que no se lograrían normalmente si todos los individuos son similares. Cuando ocurre que todos son similares o iguales (al mejor comúnmente) ocurre la situación conocida como estancamiento. Este método de selección lo evita.

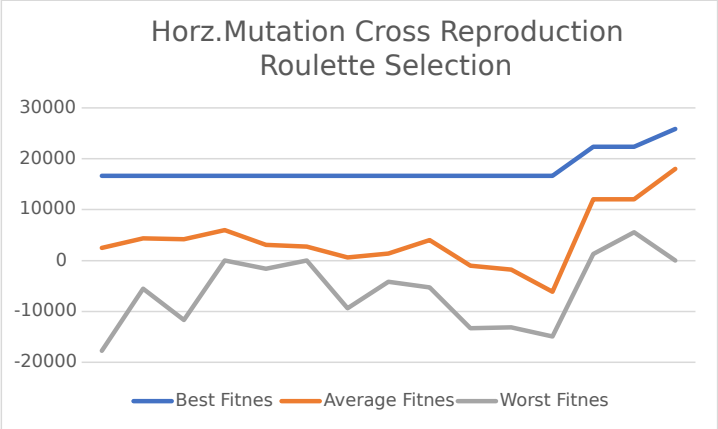

Figura 6.3: Resultados en WarriorDefense | Horizontal Mutation | ColorCross Reproduction | Roulette.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.3 con los parámetros:

- Método de mutación: *Horizontal*.
- Método de reproducción: *ColorCross*.
- Método de selección: *Roulette*.

En esta gráfica se puede observar una mejoría al final de la gráfica. La variación que se observa en la media y el peor individuo de la generación a lo largo de la ejecución son provocados por un peor o mejor rendimiento del mismo individuo en diferentes ejecuciones, sumado a la generación de nuevos individuos diferentes por los métodos de cruce y mutación dados. El funcionamiento con *roulette* y *deterministic tournament* es similar causando que la media y el peor tengan movimientos similares constantemente.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.4 con los parámetros:

- Método de mutación: *Horizontal*.
- Método de reproducción: *EvenCross*.
- Método de selección: *Deterministic tournament*.

Aunque esta gráfica solo sube, es debido a que los resultados extremos en los valores para el *Warrior Defense* obtienen buenas puntuaciones. El Método de reproducción *EvenCross* fuerza demasiado a los resultados a ir hacia los extremos, si se quiere usar eficientemente ese Método es mejor usarlo con probabilidades mucho menores.

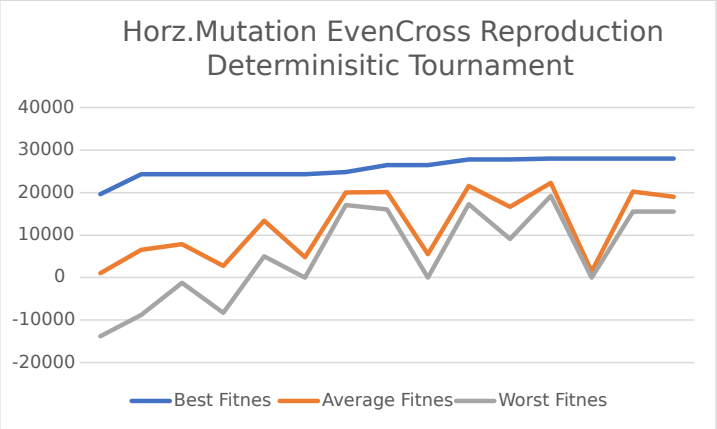

Figura 6.4: Resultados en WarriorDefense | Horizontal Mutation | EvenCross Reproduction | Deterministic tournament.

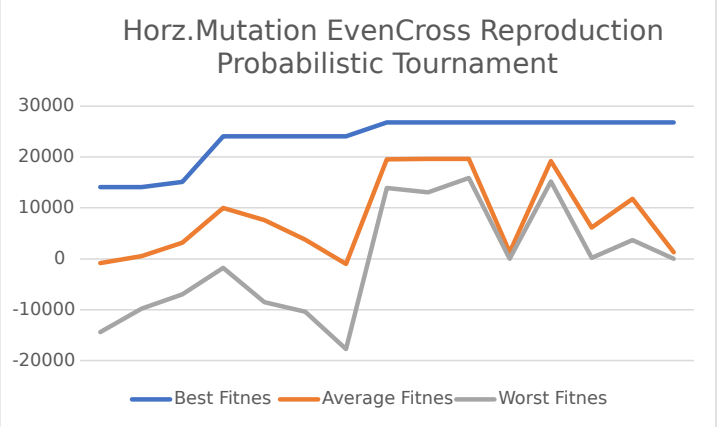

Figura 6.5: Resultados en WarriorDefense | Horizontal Mutation | EvenCross Reproduction | Probabilistic tournament.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.5 con los parámetros:

- Método de mutación: *Horizontal*.
- Método de reproducción: *EvenCross*.
- Método de selección: *Probabilistic tournament*.

Esta gráfica sufre del mismo problema que la gráfica 6.4, ya que tanto el método de reproducción como el de mutación fuerzan a la gráfica a los extremos, en el caso de la mutación es mucho más reducido esto. La diferencia principal con la gráfica 6.4 es el método de selección que permite tener más

variedad de individuos al mantener alguno de los peores entre ellos, bajando así la media y el peor individuo.

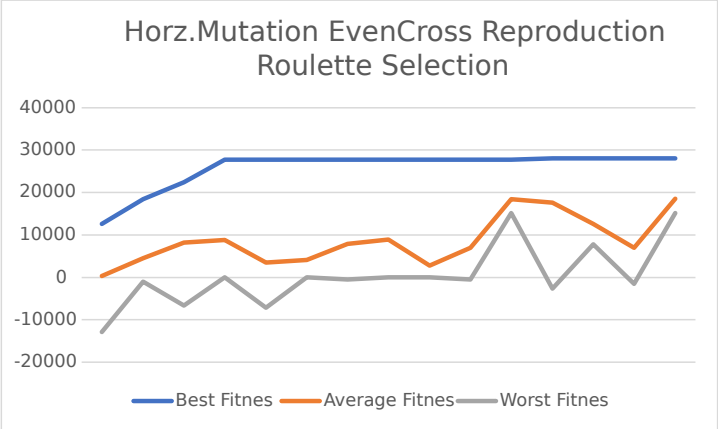

Figura 6.6: Resultados en WarriorDefense | Horizontal Mutation | EvenCross Reproduction | Roulette.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.6 con los parámetros:

- Método de mutación: *Horizontal*.
- Método de reproducción: *EvenCross*.
- Método de selección: *Roulette*.

Esta gráfica sufre del mismo problema que las gráficas 6.5 y 6.4,siendo forzada a un resultado por la mutación y el tipo de cruce. Es más similar a la que tiene el método de selección *deterministic tournament*, debido a la similitud entre estos métodos en la obtención de los mejores individuos.

Resultados obtenidos de ejecutar nuestra aplicación en 6.7 con los parámetros:

- Método de mutación: *Horizontal*.
- Método de reproducción: *SwitchCross*.
- Método de selección: *Deterministic tournament*.

Esta gráfica tiene cambios bruscos entre los individuos a causa del método de reproducción, que obtuvo una buena o mala combinación entre múltiples individuos. Este método no fuerza a los resultados a una tendencia, a cambio pierde mucha variabilidad y la capacidad de generar nuevos resultados, con individuos similares. Suele obtener buenos resultados con ejecuciones más longevas.

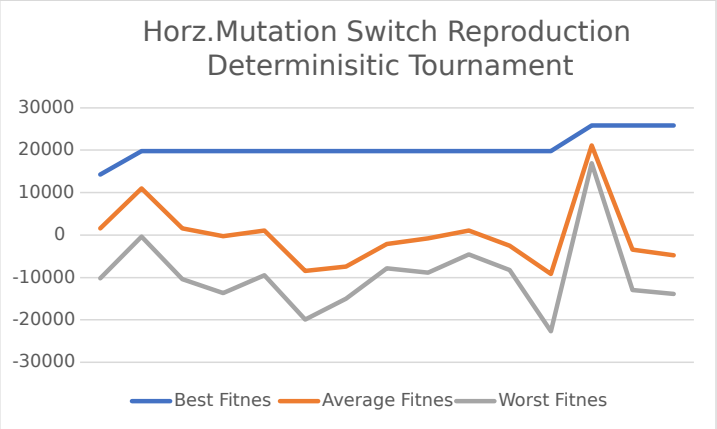

Figura 6.7: Resultados en WarriorDefense | Horizontal Mutation | Switch-Cross Reproduction | Deterministic tournament.

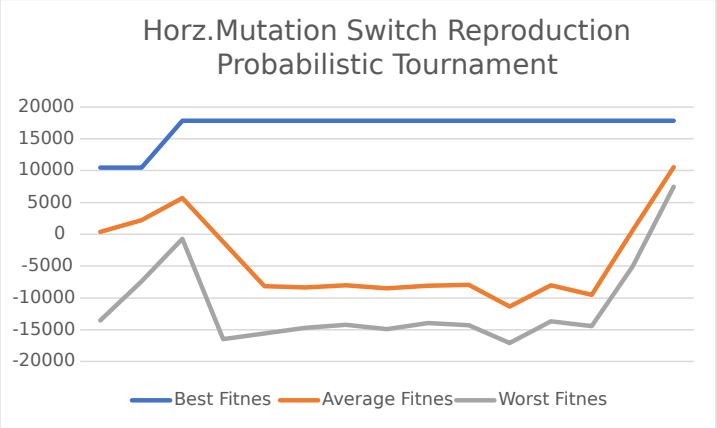

Figura 6.8: Resultados en WarriorDefense | Horizontal Mutation | Switch-Cross Reproduction | Probabilistic tournament.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.8 con los parámetros:

- Método de mutación: *Horizontal*.
- Método de reproducción: *SwitchCross*.
- Método de selección: *Probabilistic tournament*.

Esta gráfica tiene cambios bruscos entre los individuos a causa del método de reproducción, que obtuvo una buena o mala combinación entre múltiples individuos. A diferencia de la gráfica 6.7, una mejora en los resultados es más probable gracias al método de selección, este combina especialmente bien con el de selección, gracias a la variedad de individuos que genera.

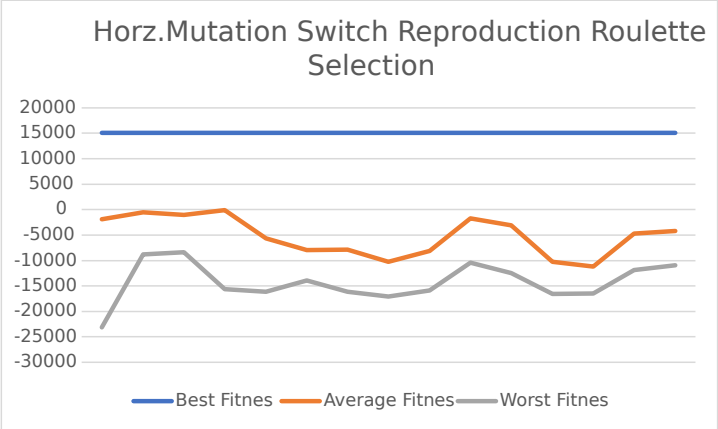

Figura 6.9: Resultados en WarriorDefense | Horizontal Mutation | Switch-Cross Reproduction | Roulette.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.9 con los parámetros:

- Método de mutación: *Horizontal*.
- Método de reproducción: *SwitchCross*.
- Método de selección: *Roulette*.

Este es un mal caso provocado por la falta de variación entre los individuo, en el cual el método de reproducción no ayuda a la mejora de estos, y el método de selección obliga a que que se estanque en ciertos resultados constantemente. Para poder mejorar los resultados obtenidos se necesitaría un método de mutación con una mayor capacidad de generar individuos aleatorios,una mayor probabilidad de mutación y un prolongado tiempo de ejecución.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.10 con los parámetros:

- Método de mutación: *Proportional*.
- Método de reproducción: *ColorCross*.
- Método de selección: *Deterministic tournament*.

En esta gráfica se aprecia un buen crecimiento y buenos resultados. Estos resultados son causados por la combinación de los métodos de mutación y cruce. Este método de mutación fuerza a generar valores extremos, lo cual da buenos resultados únicamente en este juego. A pesar de generar valores extremos, no restringe tanto como el método de reproducción *EvenCross*.

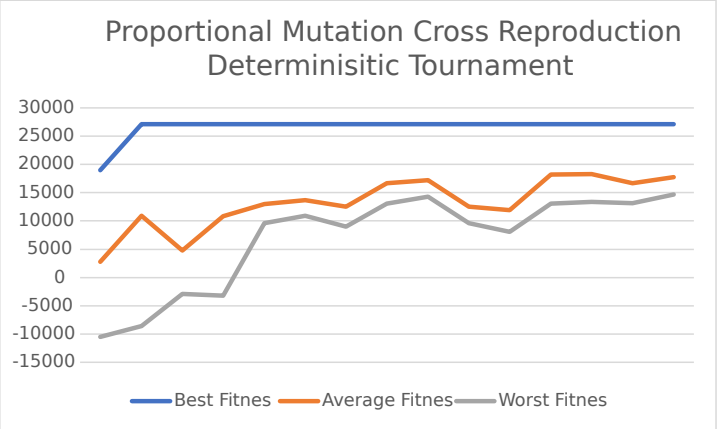

Figura 6.10: Resultados en WarriorDefense | Proportional Mutation | Color-Cross Reproduction | Deterministic tournament.

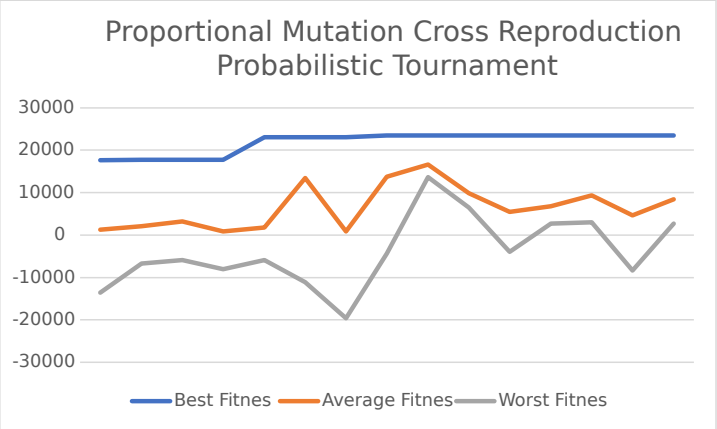

Figura 6.11: Resultados en WarriorDefense | Proportional Mutation | Color-Cross Reproduction | Probabilistic tournament.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.11 con los parámetros:

- Método de mutación: *Proportional*.
- Método de reproducción: *ColorCross*.
- Método de selección: *Probabilistic tournament*.

Esta gráfica sufre de lo mismo que la gráfica 6.10 pero en cambio no siempre son elegidos los mejores individuos en el método de selección generando esos picos en el peor individuo de cada generación, y una mayor distancia entre la media y el peor individuo.

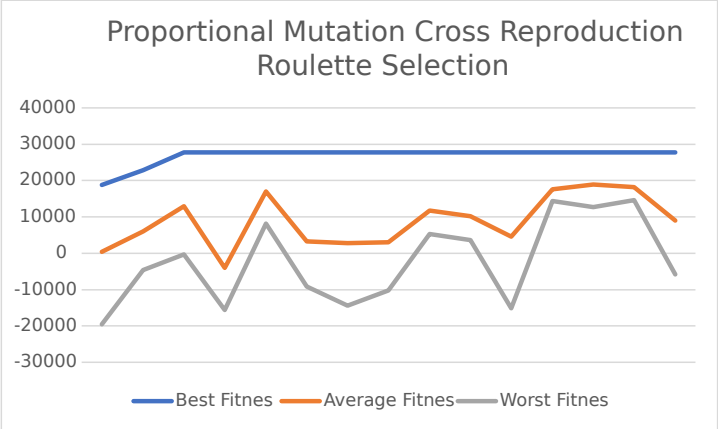

Figura 6.12: Resultados en WarriorDefense | Proportional Mutation | Color-Cross Reproduction | Roulette.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.12 con los parámetros:

- Método de mutación: *Proportional*.
- Método de reproducción: *ColorCross*.
- Método de selección: *Roulette*.

En esta gráfica se observa un crecimiento inicial muy acentuado a partir del cual la media y el peor individuo empiezan a fluctuar muy rápidamente. Esto es causado por el estancamiento de los resultados y la variación provocada por los métodos de cruce y mutación. Se recuerda que los resultados pueden estar estancados, a pesar de la gran diferencia entre el mejor individuo y la media, a causa del propio azar del juego.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.13 con los parámetros:

- Método de mutación: *Proportional*.
- Método de reproducción: *EvenCross*.
- Método de selección: *Deterministic tournament*.

En esta gráfica hay un crecimiento inicial extremo y se estabiliza inmediatamente tras pocas generaciones. Esto es causado por la combinación del método de reproducción y de mutación, obligando a los resultados a tener valores en los extremos permitidos. Son buenos resultados solo en este juego, ya que las mejores decisiones se encuentran entre esos valores.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.14 con los parámetros:

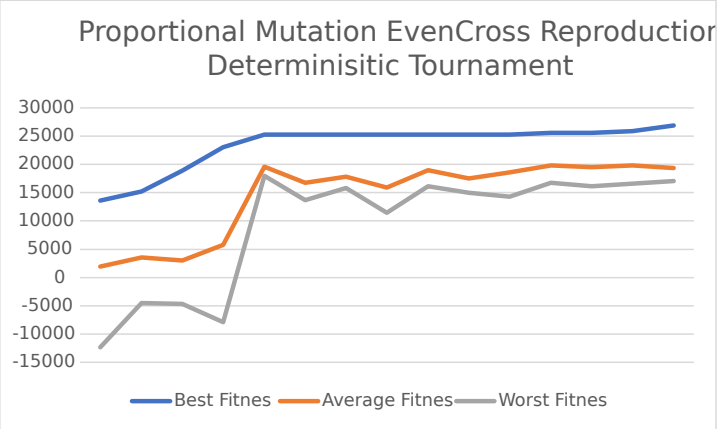

Figura 6.13: Resultados en WarriorDefense | Proportional Mutation | EvenCross Reproduction | Deterministic tournament.

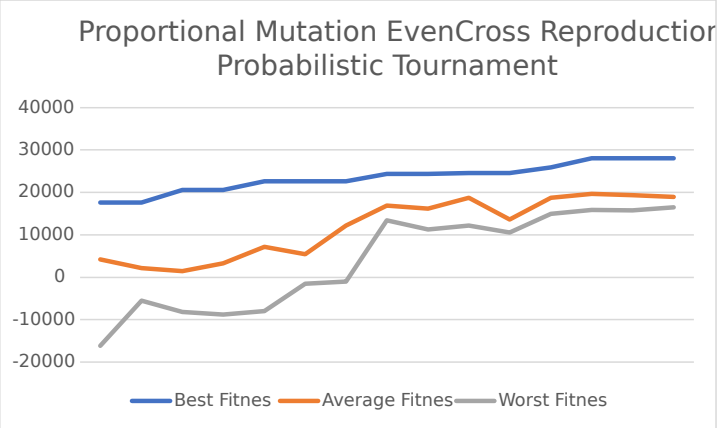

Figura 6.14: Resultados en WarriorDefense | Proportional Mutation | EvenCross Reproduction | Probabilistic tournament.

- Método de mutación: *Proportional*.
- Método de reproducción: *EvenCross*.
- Método de selección: *Probabilistic tournament*.

Esta gráfica sufre de lo mismo de la gráfica 6.13, crecimiento rápido que se estanca a causa de la combinación de los métodos de reproducción y mutación. Aunque a diferencia de la gráfica 6.13, el método de selección elegido evita ese estancamiento al menos durante más tiempo, ralentizando así también el aumento extremo en la puntuación del mejor individuo.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.15 con los parámetros:

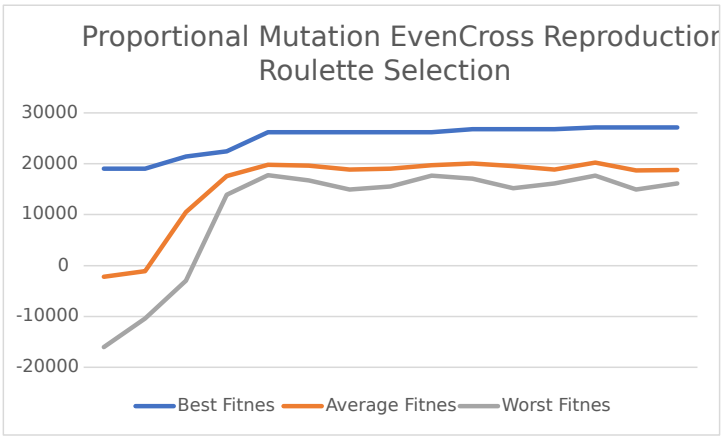

Figura 6.15: Resultados en WarriorDefense | Proportional Mutation | Even-Cross Reproduction | Roulette.

- Método de mutación: *Proportional*.
- Método de reproducción: *EvenCross*.
- Método de selección: *Roulette* .

Esta gráfica sufre de lo mismo que las gráficas 6.14 y 6.13 , crecimiento rápido que se estanca a causa de la combinación de los métodos de reproducción y mutación. Por culpa del método de selección este proceso de estancamiento se acelera muchísimo.

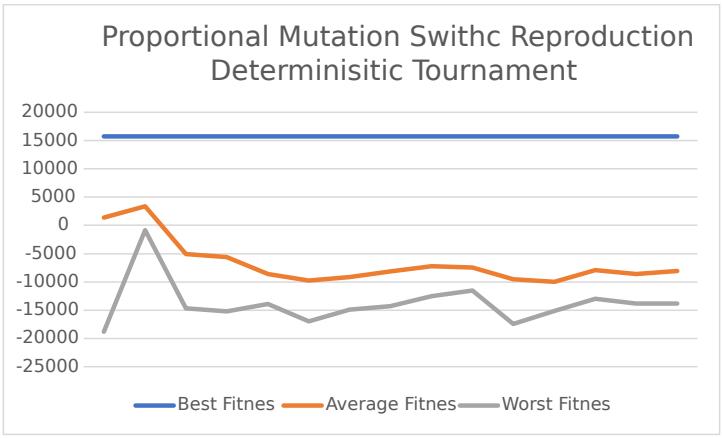

Figura 6.16: Resultados en WarriorDefense | Proportional Mutation | Switch-Cross Reproduction | Deterministic tournament.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.16 con los parámetros:

- Método de mutación: *Proportional*.
- Método de reproducción: *SwitchCross*.
- Método de selección: *Deterministic tournament*.

En este caso la gráfica tiene el comportamiento contrario a las anteriores a causa de que el método de reproducción *SwitchCross* no varía ni genera nuevos valores para los individuos. Esto en combinación con un método de mutación que fuerza los extremos no consigue lograr una mejoría. Provocando que en este juego el tiempo de decisión para sacar unidades sea extremadamente alto, haciendo así ineficiente el sacar las unidades correctas. Y por lo tanto genera esa caída en los resultados.

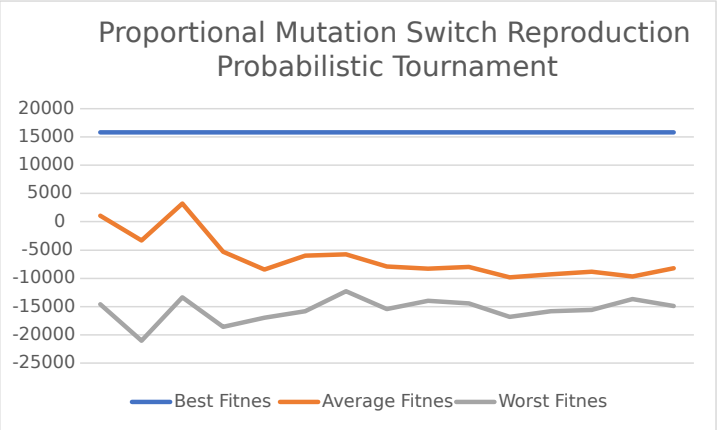

Figura 6.17: Resultados en WarriorDefense | Proportional Mutation | Switch-Cross Reproduction | Probabilistic tournament.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.17 con los parámetros:

- Método de mutación: *Proportional*.
- Método de reproducción: *SwitchCross*.
- Método de selección: *Probabilistic tournament*.

Los resultados en este ejemplo sufren de lo mismo que en el caso 6.16. En este caso el método de selección no ayuda a mejorar la gráfica, solo ralentiza ligeramente el ritmo al que baja la media.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.18 con los parámetros:

- Método de mutación: *Proportional*.

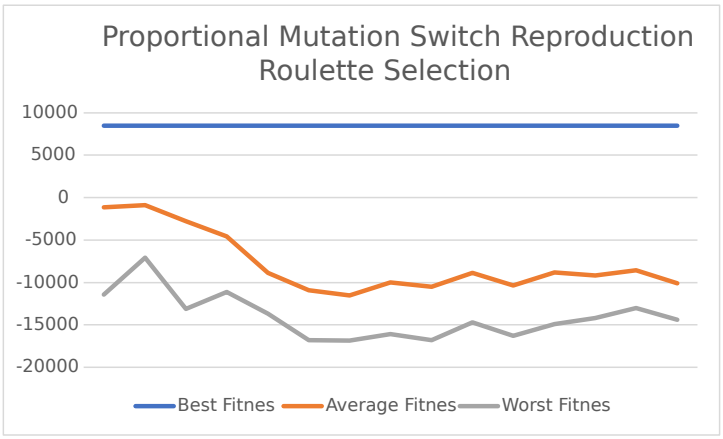

Figura 6.18: Resultados en WarriorDefense | Proportional Mutation | Switch-Cross Reproduction | Roulette.

- Método de reproducción: *SwitchCross*.
- Método de selección: *Roulette*.

En esta gráfica se observa que ocurre exactamente lo mismo que en la primera con esta combinación de métodos de reproducción y mutación. Obteniendo una media ligeramente mayor al caso mencionado a causa del método de selección.

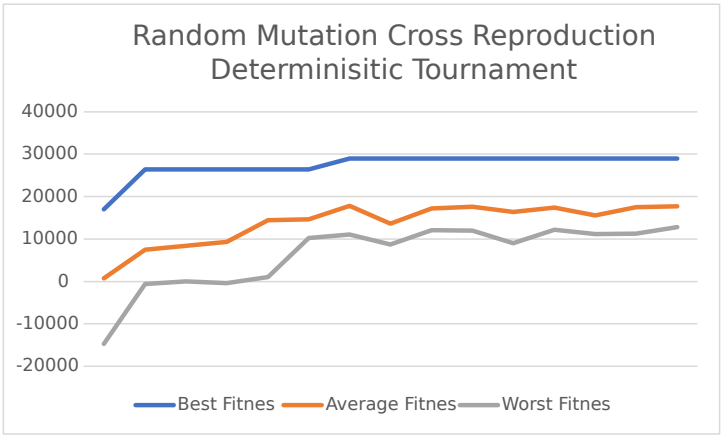

Figura 6.19: Resultados en WarriorDefense | Random Mutation | ColorCross Reproduction | Deterministic tournament.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.19 con los parámetros:

- Método de mutación: *Random*.

- Método de reproducción: *ColorCross*.
- Método de selección: *Deterministic tournament*.

Esta gráfica representa los resultados obtenidos con una de las mejores combinaciones de métodos, considerándola así a causa de no obligar ni restringir de ninguna forma los resultados. Se observa un crecimiento suave y constante, el cual se prevé que sigue mejorando cuanto mayor tiempo de ejecución hasta el punto de estancarse al encontrar la mejor solución posible.

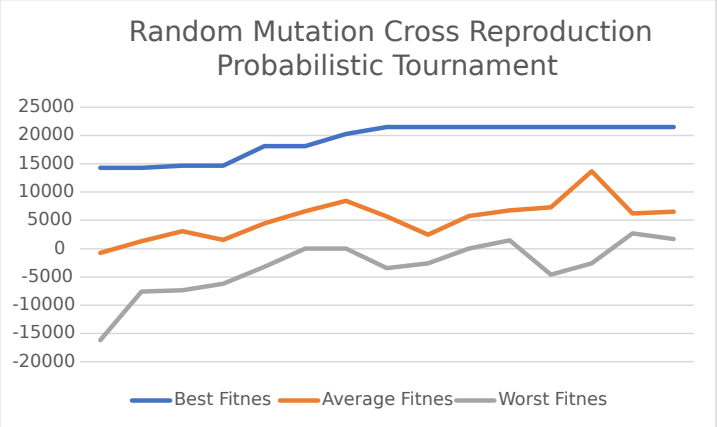

Figura 6.20: Resultados en WarriorDefense | Random Mutation | ColorCross Reproduction | Probabilistic tournament.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.20 con los parámetros:

- Método de mutación: *Random*.
- Método de reproducción: *ColorCross*.
- Método de selección: *Probabilistic tournament*.

Esta gráfica representa los resultados obtenidos con una de las mejores combinaciones de métodos, considerándola así a causa de no obligar ni restringir de ninguna forma los resultados. Se observa un crecimiento suave y constante, el cual se prevé que sigue mejorando cuanto mayor tiempo de ejecución hasta el punto de estancarse al encontrar la mejor solución posible. Esta combinación a diferencia de la anterior es algo más lenta en obtener buenos resultados, pero asegura que no se estanque nunca el algoritmo. Esta combinación permite encontrar de forma segura el mejor valor posible entre todos a costa de un tiempo de ejecución mucho mayor.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.21 con los parámetros:

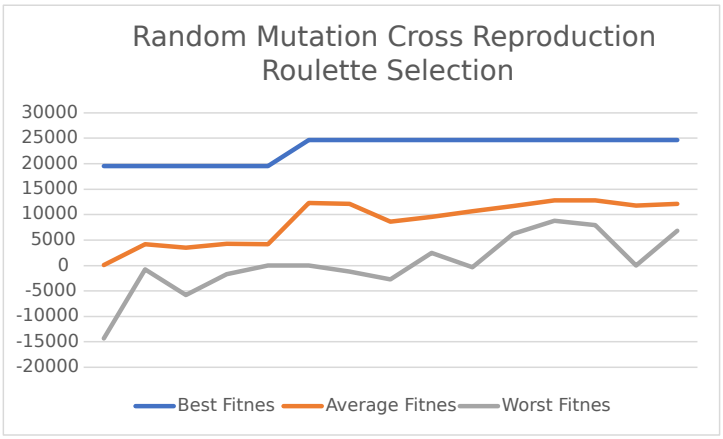

Figura 6.21: Resultados en WarriorDefense | Random Mutation | ColorCross Reproduction | Roulette.

- Método de mutación: *Random*.
- Método de reproducción: *ColorCross*.
- Método de selección: *Roulette* .

Esta gráfica representa los resultados obtenidos con una de las mejores combinaciones de métodos, considerándola así a causa de no obligar ni restringir de ninguna forma los resultados. Se observa un crecimiento suave y constante, el cual se prevé que sigue mejorando cuanto mayor tiempo de ejecución hasta el punto de estancarse al encontrar la mejor solución posible.

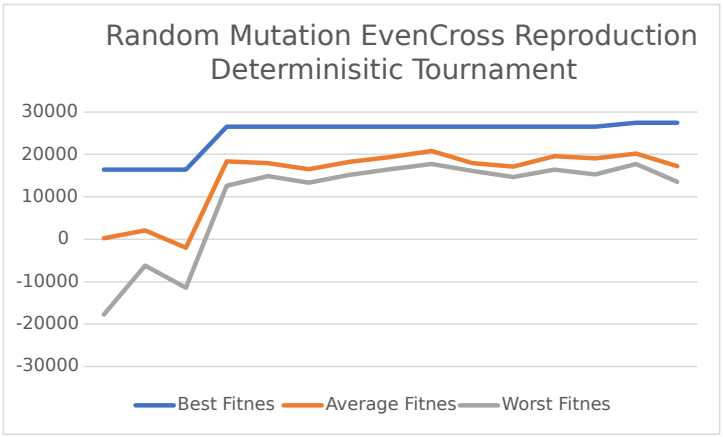

Figura 6.22: Resultados en WarriorDefense | Random Mutation | EvenCross Reproduction | Deterministic tournament.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.22 con

los parámetros:

- Método de mutación: *Random*.
- Método de reproducción: *EvenCross*.
- Método de selección: *Deterministic tournament*.

Esta gráfica se estanca inmediatamente a causa del método de reproducción. Los resultados obtenidos están completamente condicionados por el método de reproducción.

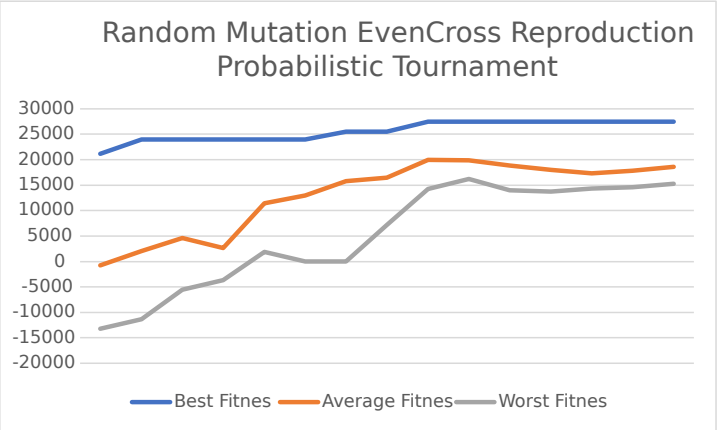

Figura 6.23: Resultados en WarriorDefense | Random Mutation | EvenCross Reproduction | Probabilistic tournament.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.23 con los parámetros:

- Método de mutación: *Random*.
- Método de reproducción: *EvenCross*.
- Método de selección: *Probabilistic tournament*.

Esta gráfica se estanca rápidamente, aunque no tanto como en el caso 6.22 gracias al método de selección que permite tener más individuos diferentes. Los resultados obtenidos están completamente condicionados por el método de reproducción.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.24 con los parámetros:

- Método de mutación: *Random*.
- Método de reproducción: *EvenCross*.

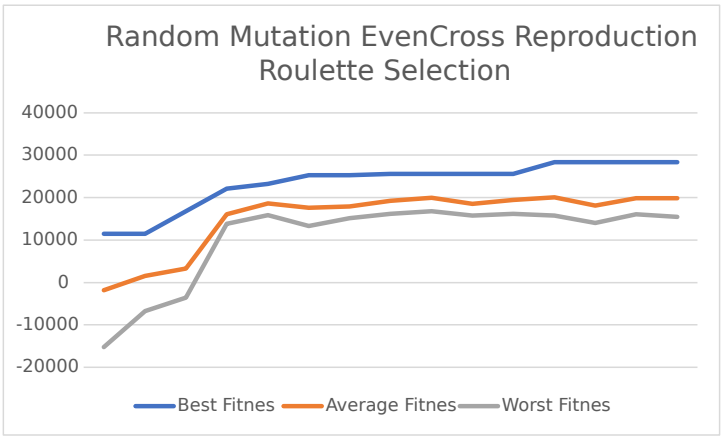

Figura 6.24: Resultados en WarriorDefense | Random Mutation | EvenCross Reproduction | Roulette.

- Método de selección: *Roulette*.

Esta gráfica se estanca inmediatamente a causa del método de reproducción. Los resultados obtenidos están completamente condicionados por el método de reproducción.

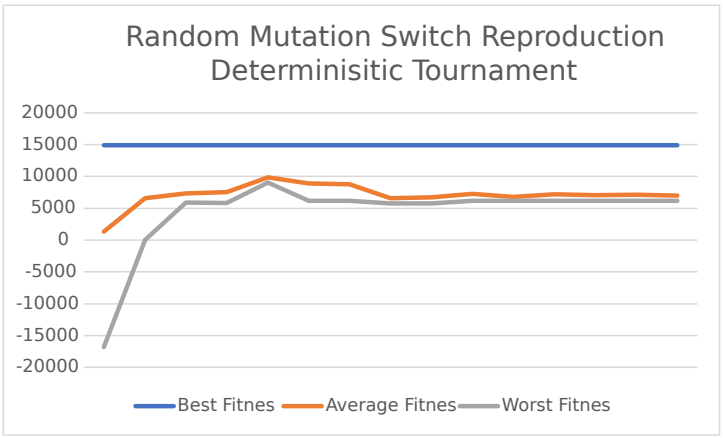

Figura 6.25: Resultados en WarriorDefense | Random Mutation | SwitchCross Reproduction | Deterministic tournament.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.25 con los parámetros:

- Método de mutación: *Random*.
- Método de reproducción: *SwitchCross*.
- Método de selección: *Deterministic tournament*.

Los resultados de esta gráfica son tan extraños debido a la poca presencia que tiene el método de reproducción en ellos. Generando poca variabilidad. Por lo tanto la gráfica se estanca inmediatamente a causa del método de selección.

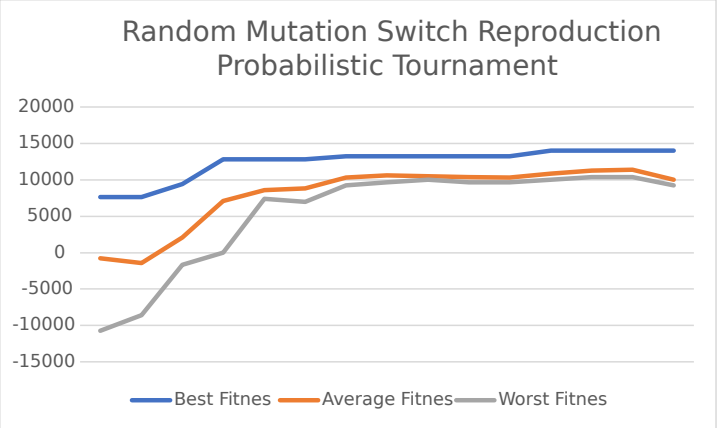

Figura 6.26: Resultados en WarriorDefense | Random Mutation | SwitchCross Reproduction | Probabilistic tournament.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.26 con los parámetros:

- Método de mutación: *Random*.
- Método de reproducción: *SwitchCross*.
- Método de selección: *Probabilistic tournament*.

Los resultados de esta gráfica son tan extraños debido a la poca presencia que tiene el método de reproducción en ellos. Generando poca variabilidad. Por lo tanto la gráfica se estanca inmediatamente a causa del método de selección. A diferencia de la gráfica 6.25 en esta se consigue lograr ciertas mejoras durante la ejecución gracias a la variedad de individuos extra otorgada por el método de selección.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.27 con los parámetros:

- Método de mutación: *Random*.
- Método de reproducción: *SwitchCross*.
- Método de selección: *Roulette*.

Los resultados de esta gráfica son debido a la poca presencia que tiene el método de reproducción en ellos. Generando poca variabilidad. Por lo tanto

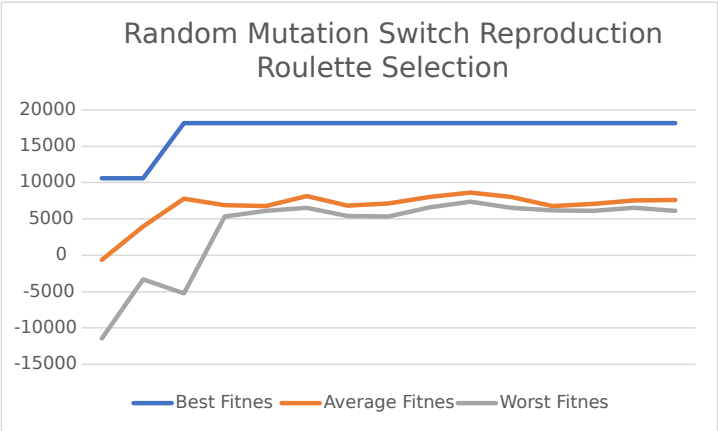

Figura 6.27: Resultados en WarriorDefense | Random Mutation | SwitchCross Reproduction | Roulette.

la gráfica se estanca inmediatamente a causa del método de selección. Aunque no parezca estancada por la distancia que hay entre la mejor puntuación y la media, esa distancia es provocada por una variación en los resultados causada por el azar propio del juego.

6.1.2. *MicroRTS*

Las pruebas realizadas en el juego RTS tienen como parametros en común:

- Tiempo de juego: 60 segundos.
- Número de generaciones: 20 generaciones o iteraciones del algoritmo.
- Velocidad del juego: 2 veces más rápido.(No está más rápido para no estropear el funcionamiento del juego.)
- Tamaño de la población: 15 individuos.(Una población más pequeña debido a la mayor complejidad del juego y no muy buena optimización de este.
- Porcentaje de elitismo: 15 %.(Al menos 2 individuos)
- Probabilidad de mutación: 30 %.
- Probabilidad de reproducción: 60 %.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.28 con los parámetros:

- Método de mutación: *Horizontal*.

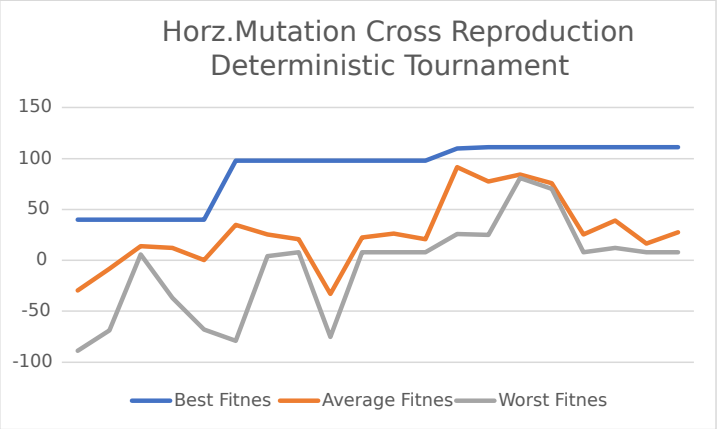

Figura 6.28: Resultados en MicroRTS | Horizontal Mutation | ColorCross Reproduction | Deterministic tournament.

- Método de reproducción: *ColorCross*.
- Método de selección: *Deterministic tournament*.

En esta gráfica se puede observar una mejoría a lo largo de la ejecución hasta llegar a un máximo. La variación tan extrema que se observa a veces en la media y el peor individuo de la generación son provocados por un peor o mejor rendimiento del mismo individuo en diferentes ejecuciones, sumado a la generación de nuevos individuos diferentes por los métodos de cruce y mutación dados.

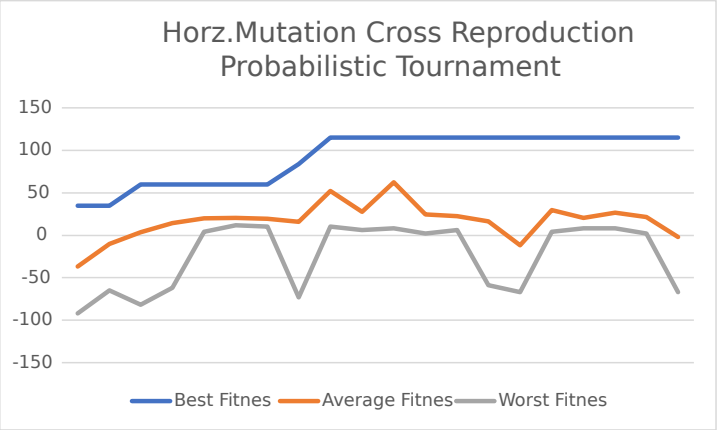

Figura 6.29: Resultados en MicroRTS | Horizontal Mutation | ColorCross Reproduction | Probabilistic tournament.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.29 con los parámetros:

- Método de mutación: *Horizontal*.
- Método de reproducción: *ColorCross*.
- Método de selección: *Probabilistic tournament*.

En esta gráfica se puede observar una mejoría a lo largo de la ejecución hasta llegar a un máximo. En cambio esta sufre de una media y un peor individuo en general inferiores en puntuación a la de la gráfica 6.28. Esto es provocado por el método de selección que favorece de vez en cuando a los peores individuos.

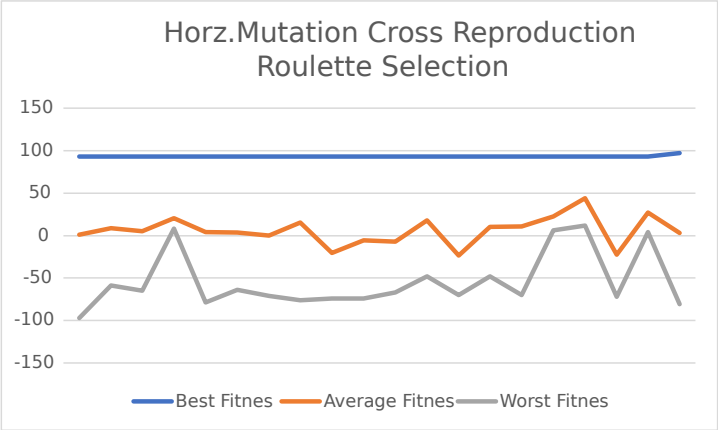

Figura 6.30: Resultados en MicroRTS | Horizontal Mutation | ColorCross Reproduction | Roulette.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.30 con los parámetros:

- Método de mutación: *Horizontal*.
- Método de reproducción: *ColorCross*.
- Método de selección: *Roulette*.

En esta gráfica no se puede observar una mejora en los resultados hasta llegar a la ultima generación debido a obtener un buen resultado inicial. A pesar de eso en el peor individuo se nota una ligera tendencia a subir en puntuación. Esta tendencia se rompe en muchos de los picos a causa de los métodos de reproducción y mutación.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.31 con los parámetros:

- Método de mutación: *Horizontal*.

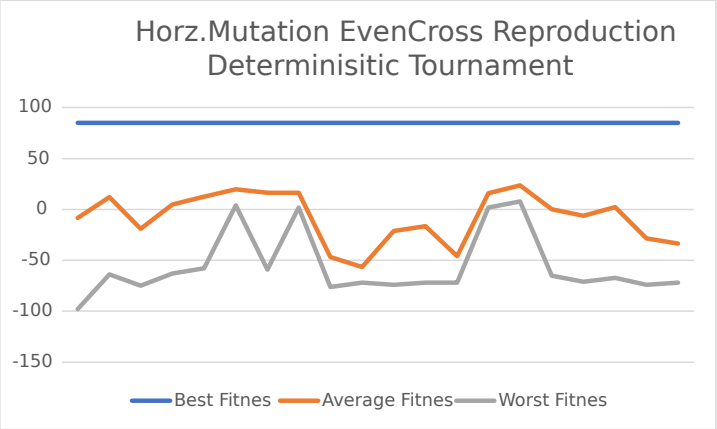

Figura 6.31: Resultados en MicroRTS | Horizontal Mutation | EvenCross Reproduction | Deterministic tournament.

- Método de reproducción: *EvenCross*.
- Método de selección: *Deterministic tournament*.

En esta gráfica se muestra que los resultados obtenidos con el método de reproducción *EvenCross* no son buenos. Como se ha mencionado anteriormente este método tiene la tendencia de forzar a los resultados a valores extremos. En este juego esos valores no dan buenos resultados. La gráfica adquiere esta forma extraña a causa de que la mutación genera de vez en cuando mejores resultados a los anteriores.

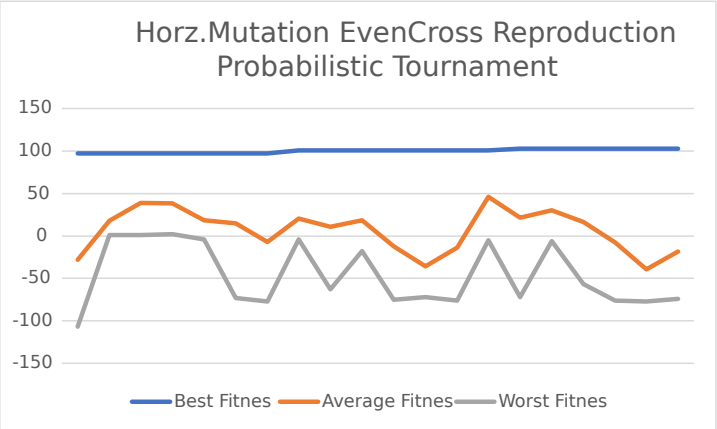

Figura 6.32: Resultados en MicroRTS | Horizontal Mutation | EvenCross Reproduction | Probabilistic tournament.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.32 con los parámetros:

- Método de mutación: *Horizontal*.
- Método de reproducción: *EvenCross*.
- Método de selección: *Probabilistic tournament*.

En esta gráfica se muestra que los resultados obtenidos con el método de reproducción *EvenCross* no son buenos. Como se ha mencionado anteriormente este método tiene la tendencia de forzar a los resultados a valores extremos. En este juego esos valores no dan buenos resultados.El método de selección escogido no ayuda a obtener mejores resultados, al permitir la persistencia de malos individuos sin un buen método de cruce para variarlos.

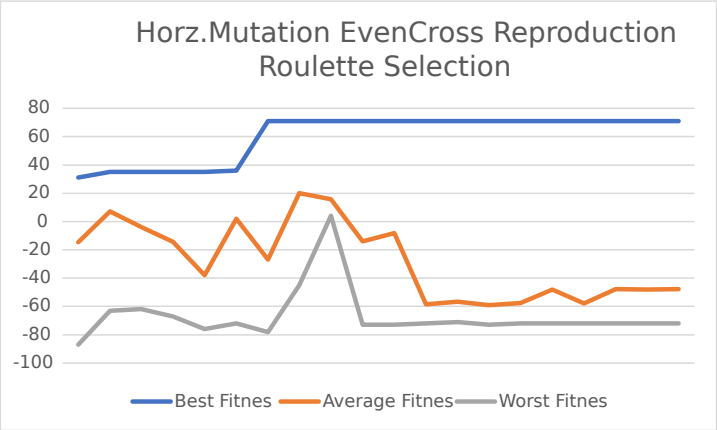

Figura 6.33: Resultados en MicroRTS | Horizontal Mutation | EvenCross Reproduction | Roulette.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.33 con los parámetros:

- Método de mutación: *Horizontal*.
- Método de reproducción: *EvenCross*.
- Método de selección: *Roulette*.

Aunque se observe una mejora en el mejor resultado a mitad de ejecución, al observar la media y el peor individuo, se comprende que no esta obteniendo resultados correctos y por lo tanto los métodos elegidos de mutación y reproducción no son adecuados.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.34 con los parámetros:

- Método de mutación: *Horizontal*.

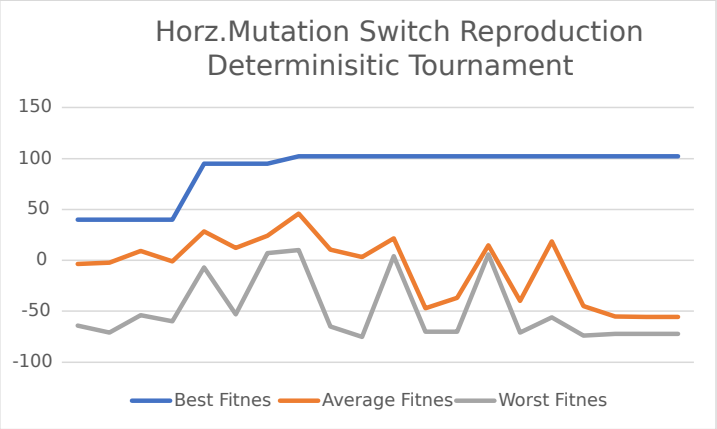

Figura 6.34: Resultados en MicroRTS | Horizontal Mutation | SwitchCross Reproduction | Deterministic tournament.

- Método de reproducción: *SwitchCross*.
- Método de selección: *Deterministic tournament*.

Esta gráfica tiene cambios bruscos entre los individuos a causa del método de reproducción, que obtuvo una buena o mala combinación entre múltiples individuos. Este método no fuerza a los resultados a una tendencia, a cambio pierde mucha variabilidad y la capacidad de generar nuevos resultados, con individuos similares. Aun así la media y el peor individuo caen a causa de que el método de mutación si fuerza ligeramente los resultados a valores localizados en los extremos, los cuales no funcionan correctamente en este juego.

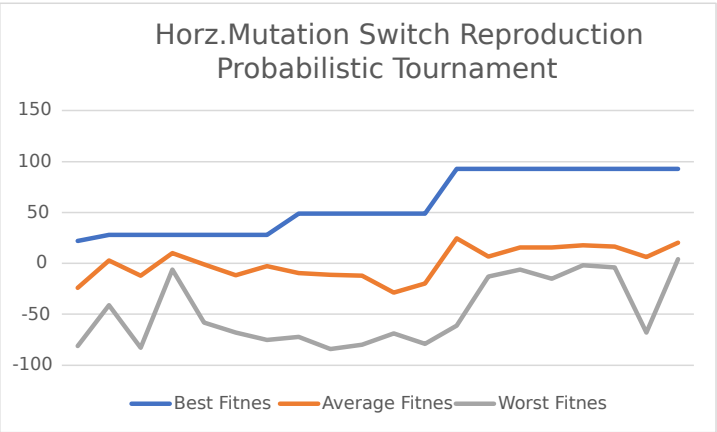

Figura 6.35: Resultados en MicroRTS | Horizontal Mutation | SwitchCross Reproduction | Probabilistic tournament.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.35 con los parámetros:

- Método de mutación: *Horizontal*.
- Método de reproducción: *SwitchCross*.
- Método de selección: *Probabilistic tournament*.

Esta gráfica tiene cambios bruscos entre los individuos a causa del método de reproducción, que obtuvo una buena o mala combinación entre múltiples individuos. A diferencia de la gráfica 6.34, una mejora en los resultados es más probable gracias al método de selección, este combina especialmente bien con el de selección, gracias a la variedad de individuos que genera.

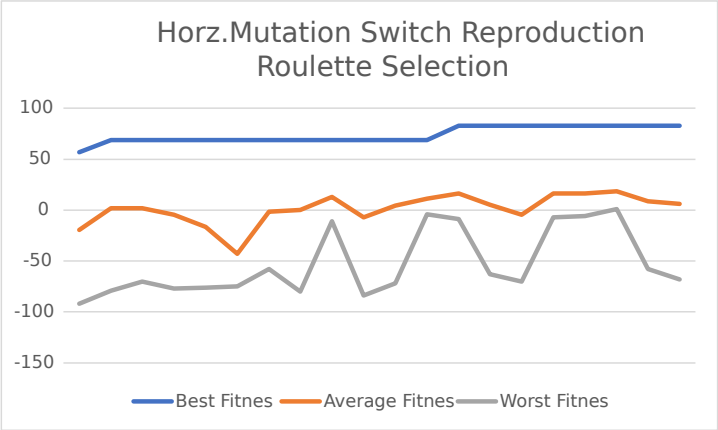

Figura 6.36: Resultados en MicroRTS | Horizontal Mutation | SwitchCross Reproduction | Roulette.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.36 con los parámetros:

- Método de mutación: *Horizontal*.
- Método de reproducción: *SwitchCross*.
- Método de selección: *Roulette*.

Esta gráfica tiene cambios bruscos entre los peores resultados a causa del método de reproducción, que obtuvo una buena o mala combinación entre múltiples individuos. El método de selección esta manteniendo los mejores resultados, evitando que la mutación estanque los individuos con malos resultados.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.37 con los parámetros:

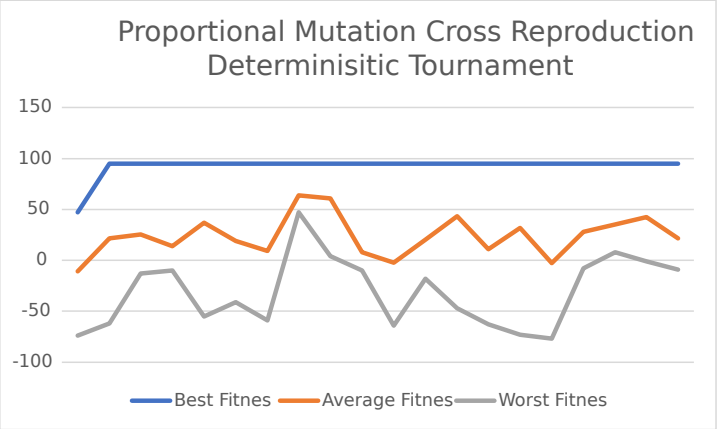

Figura 6.37: Resultados en MicroRTS | Proportional Mutation | ColorCross Reproduction | Deterministic tournament.

- Método de mutación: *Proportional*.
- Método de reproducción: *ColorCross*.
- Método de selección: *Deterministic tournament*.

A pesar de tener un método de mutación que fuerza bastante los extremos, se están obteniendo buenos resultados gracias a los métodos de reproducción y selección, y a una probabilidad más baja en la mutación que en la reproducción.

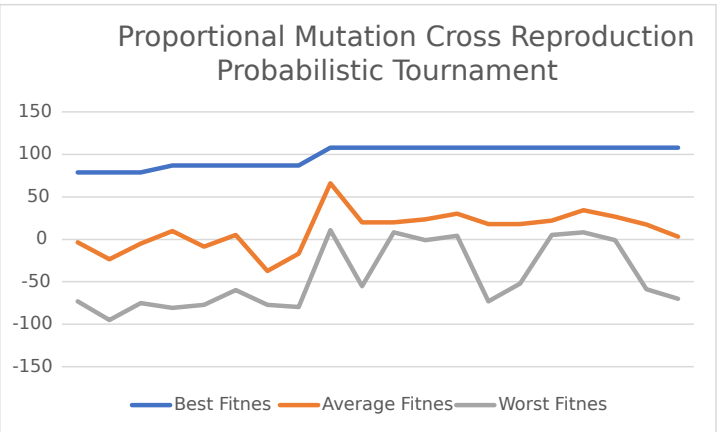

Figura 6.38: Resultados en MicroRTS | Proportional Mutation | ColorCross Reproduction | Probabilistic tournament.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.38 con los parámetros:

- Método de mutación: *Proportional*.
- Método de reproducción: *ColorCross*.
- Método de selección: *Probabilistic tournament*.

A pesar de tener un método de mutación que fuerza bastante los extremos, se están obteniendo buenos resultados gracias a los métodos de reproducción y selección, y a una probabilidad más baja en la mutación que en la reproducción. A diferencia de en el caso 6.38, la media y el peor resultado son inferiores a causa del método de selección.

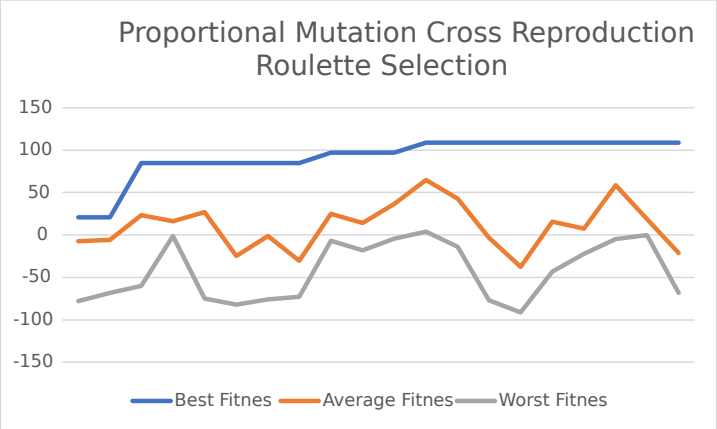

Figura 6.39: Resultados en MicroRTS | Proportional Mutation | ColorCross Reproduction | Roulette.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.39 con los parámetros:

- Método de mutación: *Proportional*.
- Método de reproducción: *ColorCross*.
- Método de selección: *Roulette*.

A pesar de tener un método de mutación que fuerza los extremos, se están obteniendo buenos resultados gracias a los métodos de reproducción y selección, y a una probabilidad más baja en la mutación que en la reproducción.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.40 con los parámetros:

- Método de mutación: *Proportional*.
- Método de reproducción: *EvenCross*.

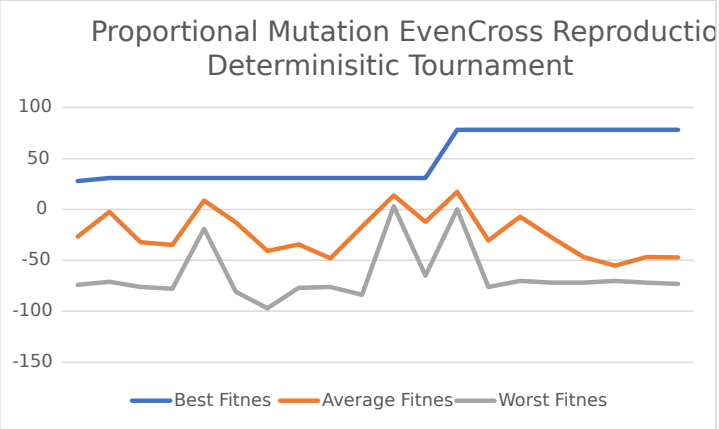

Figura 6.40: Resultados en MicroRTS | Proportional Mutation | EvenCross Reproduction | Deterministic tournament.

- Método de selección: *Deterministic tournament*.

En la gráfica se observa como los resultados simplemente empeoran en gran medida a causa de los métodos de reproducción y mutación. Ambos generando valores extremos entre los disponibles. Este tipo de valores no son adecuados para este juego provocando así la caída en la media, y una mejora muy pequeña en el mejor individuo.

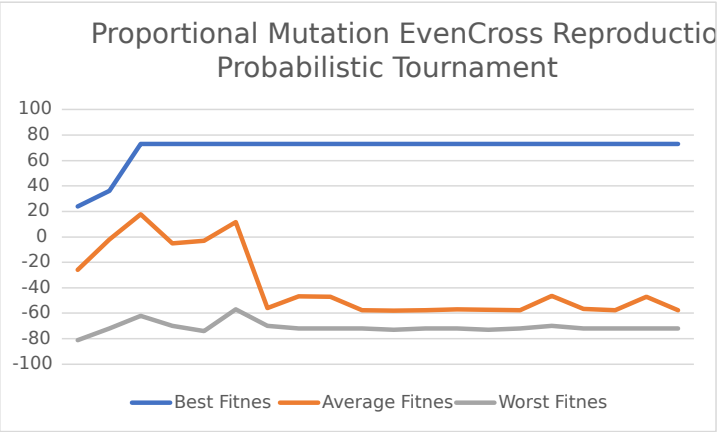

Figura 6.41: Resultados en MicroRTS | Proportional Mutation | EvenCross Reproduction | Probabilistic tournament.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.41 con los parámetros:

- Método de mutación: *Proportional*.

- Método de reproducción: *EvenCross*.
- Método de selección: *Probabilistic tournament*.

En la gráfica se observa como los resultados simplemente empeoran en gran medida a causa de los métodos de reproducción y mutación. Ambos generando valores extremos entre los disponibles. Este tipo de valores no son adecuados para este juego provocando así la caída en la media, y una mejora muy pequeña en el mejor individuo.

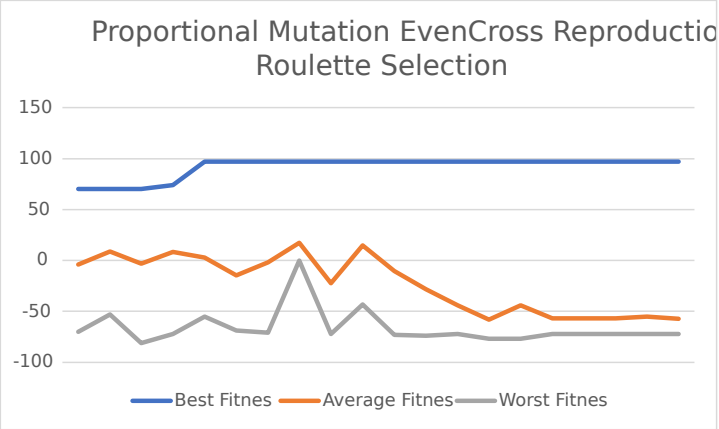

Figura 6.42: Resultados en MicroRTS | Proportional Mutation | EvenCross Reproduction | Roulette.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.42 con los parámetros:

- Método de mutación: *Proportional*.
- Método de reproducción: *EvenCross*.
- Método de selección: *Roulette*.

En la gráfica se observa como los resultados simplemente empeoran en gran medida a causa de los métodos de reproducción y mutación. Ambos generando valores extremos entre los disponibles. Este tipo de valores no son adecuados para este juego provocando así la caída en la media, y una mejora muy pequeña en el mejor individuo.

Resultados obtenidos de ejecutar nuestra aplicación en 6.43 con los parámetros:

- Método de mutación: *Proportional*.
- Método de reproducción: *SwitchCross*.

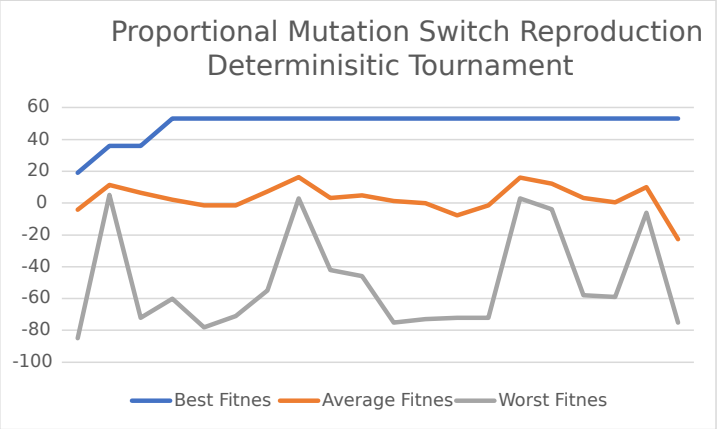

Figura 6.43: Resultados en MicroRTS | Proportional Mutation | SwitchCross Reproduction | Deterministic tournament.

- Método de selección: *Deterministic tournament*.

A causa de que el método de reproducción elegido no genera la suficiente variedad y que el método de mutación elegido favorece los valores extremos,los cuales no son adecuados para este juego, no se logra ninguna mejoría e incluso tiene una tendencia a empeorar.

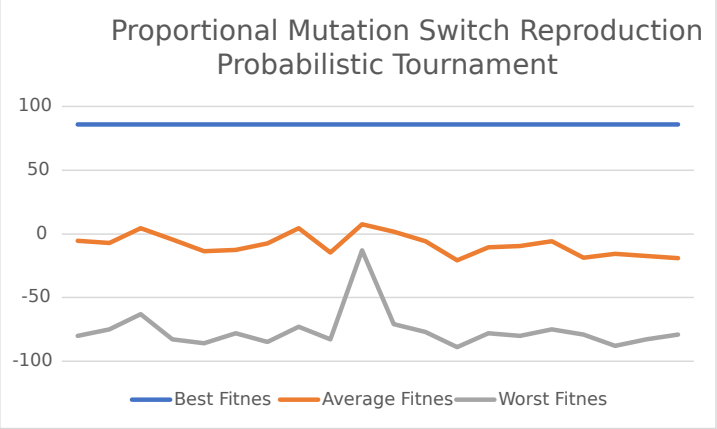

Figura 6.44: Resultados en MicroRTS | Proportional Mutation | SwitchCross Reproduction | Probabilistic tournament.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.44 con los parámetros:

- Método de mutación: *Proportional*.
- Método de reproducción: *SwitchCross*.

- Método de selección: *Probabilistic tournament*.

A causa de que el método de reproducción elegido no genera la suficiente variedad y que el método de mutación elegido favorece los valores extremos,los cuales no son adecuados para este juego, no se logra ninguna mejoría e incluso tiene una tendencia a empeorar.

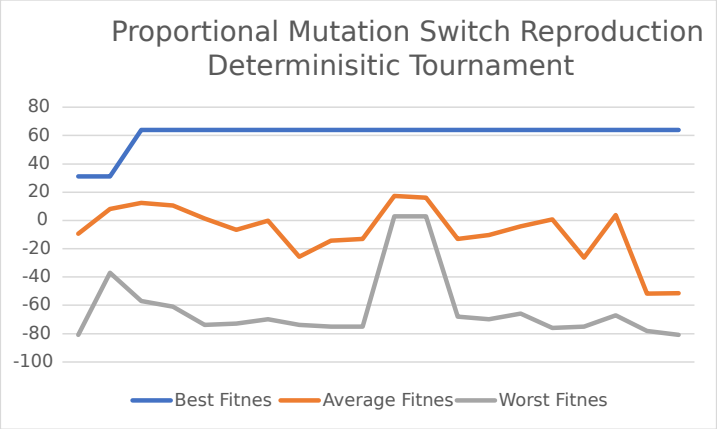

Figura 6.45: Resultados en MicroRTS | Proportional Mutation | SwitchCross Reproduction | Roulette.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.45 con los parámetros:

- Método de mutación: *Proportional*.
- Método de reproducción: *SwitchCross*.
- Método de selección: *Roulette*.

A causa de que el método de reproducción elegido no genera la suficiente variedad y que el método de mutación elegido favorece los valores extremos,los cuales no son adecuados para este juego, no se logra ninguna mejoría notable e incluso tiene una tendencia a empeorar.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.46 con los parámetros:

- Método de mutación: *Random*.
- Método de reproducción: *ColorCross*.
- Método de selección: *Deterministic tournament*.

Esta gráfica representa los resultados obtenidos con una de las mejores combinaciones de métodos, considerándola así a causa de no obligar ni restringir de ninguna forma los resultados. Se observa un crecimiento suave y

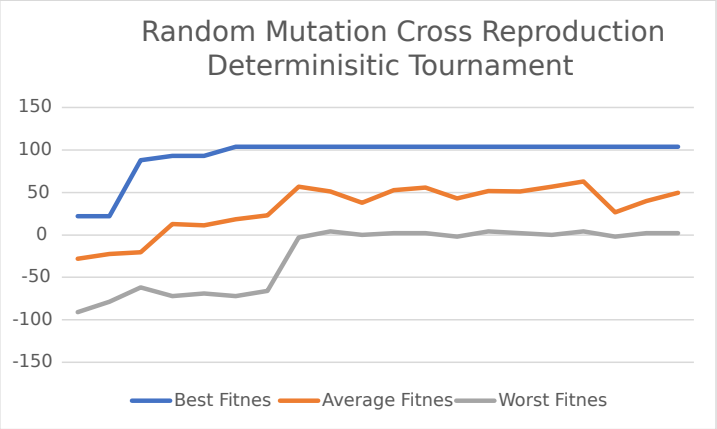

Figura 6.46: Resultados en MicroRTS | Random Mutation | ColorCross Reproduction | Deterministic tournament.

constante, el cual se prevé que sigue mejorando cuanto mayor tiempo de ejecución hasta el punto de estancarse al encontrar la mejor solución posible.

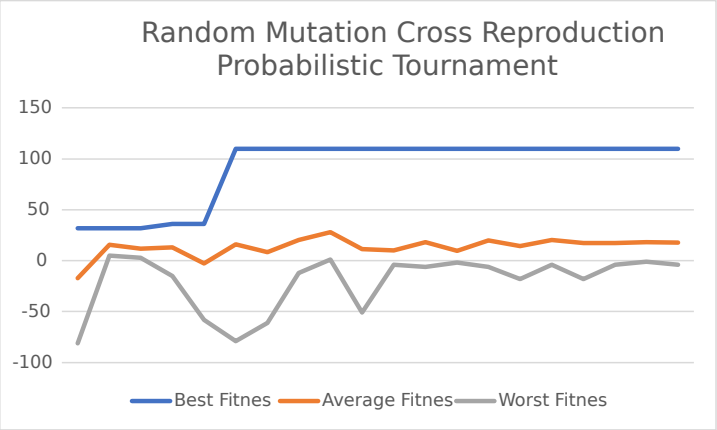

Figura 6.47: Resultados en MicroRTS | Random Mutation | ColorCross Reproduction | Probabilistic tournament.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.47 con los parámetros:

- Método de mutación: *Random*.
- Método de reproducción: *ColorCross*.
- Método de selección: *Probabilistic tournament*.

Esta gráfica representa los resultados obtenidos con una de las mejores combinaciones de métodos, considerándola así a causa de no obligar ni res-

tringir de ninguna forma los resultados. Se observa un crecimiento suave y constante, el cual se prevé que sigue mejorando cuanto mayor tiempo de ejecución hasta el punto de estancarse al encontrar la mejor solución posible. Esta combinación a diferencia de en el caso 6.46 es algo más lenta en obtener buenos resultados, pero asegura que no se estanque nunca el algoritmo. Esta combinación permite encontrar de forma segura el mejor valor posible entre todos a costa de un tiempo de ejecución mucho mayor.

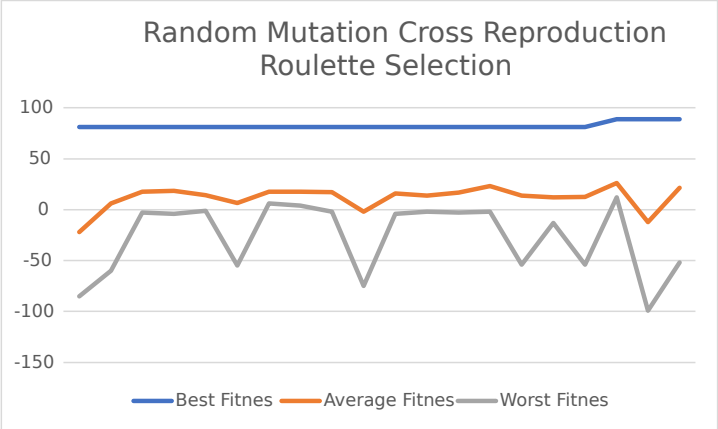

Figura 6.48: Resultados en MicroRTS | Random Mutation | ColorCross Reproduction | Roulette.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.48 con los parámetros:

- Método de mutación: *Random*.
- Método de reproducción: *ColorCross*.
- Método de selección: *Roulette*.

Esta gráfica representa los resultados obtenidos con una de las mejores combinaciones de métodos, considerándola así a causa de no obligar ni restringir de ninguna forma los resultados. Se observa un crecimiento suave y constante, el cual se prevé que sigue mejorando cuanto mayor tiempo de ejecución hasta el punto de estancarse al encontrar la mejor solución posible.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.49 con los parámetros:

- Método de mutación: *Random*.
- Método de reproducción: *EvenCross*.
- Método de selección: *Deterministic tournament*.

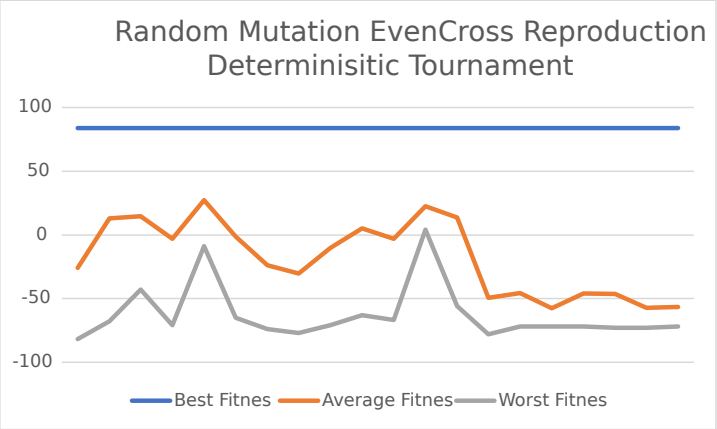

Figura 6.49: Resultados en MicroRTS | Random Mutation | EvenCross Reproduction | Deterministic tournament.

En este caso a pesar de que empiece con un buen valor inicial, la media cae inmediatamente a valores muy bajos. Todo esto es ocasionado por el método de reproducción.

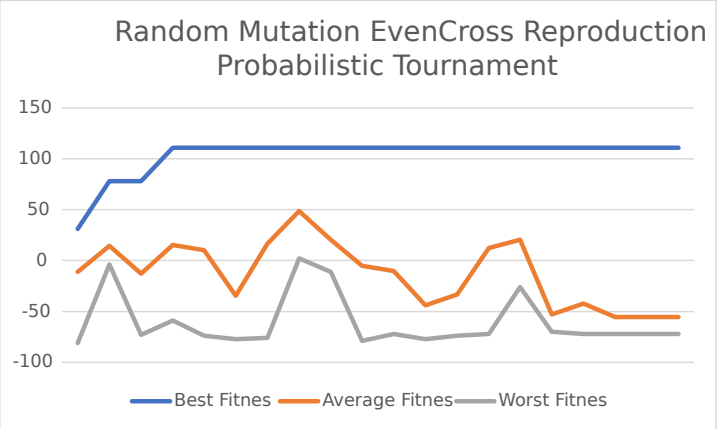

Figura 6.50: Resultados en MicroRTS | Random Mutation | EvenCross Reproduction | Probabilistic tournament.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.50 con los parámetros:

- Método de mutación: *Random*.
- Método de reproducción: *EvenCross*.
- Método de selección: *Probabilistic tournament*.

En este caso a pesar de que es capaz de mejorar en el inicio debido a la variedad generada por el método de selección y el de mutación, la media cae a valores muy bajos. Todo esto es ocasionado por el método de reproducción.

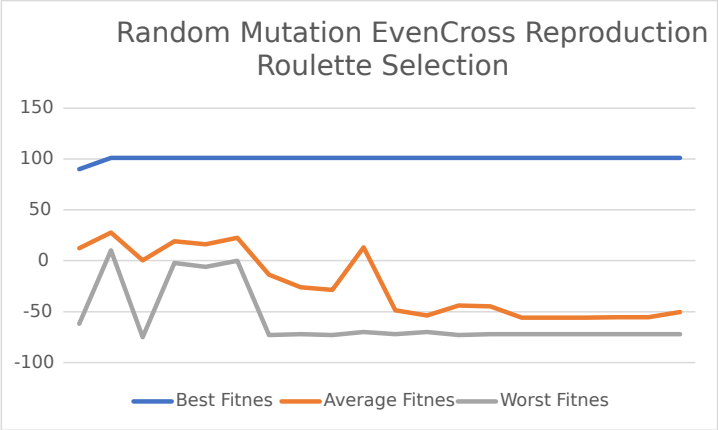

Figura 6.51: Resultados en MicroRTS | Random Mutation | EvenCross Reproduction | Roulette.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.51 con los parámetros:

- Método de mutación: *Random*.
- Método de reproducción: *EvenCross*.
- Método de selección: *Roulette*.

En este caso a pesar de que empiece con un buen valor inicial, la media cae inmediatamente a valores muy bajos. Todo esto es ocasionado por el método de reproducción.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.52 con los parámetros:

- Método de mutación: *Random*.
- Método de reproducción: *SwitchCross*.
- Método de selección: *Deterministic tournament*.

Los resultados generados son tan malos a causa de la poca variedad generada por el método de reproducción *SwitchCross*, no obteniendo así una mejoría en los resultados. Para obtener una mejoría habría que aumentar el numero de iteraciones realizadas en la ejecución y la probabilidad de mutación.

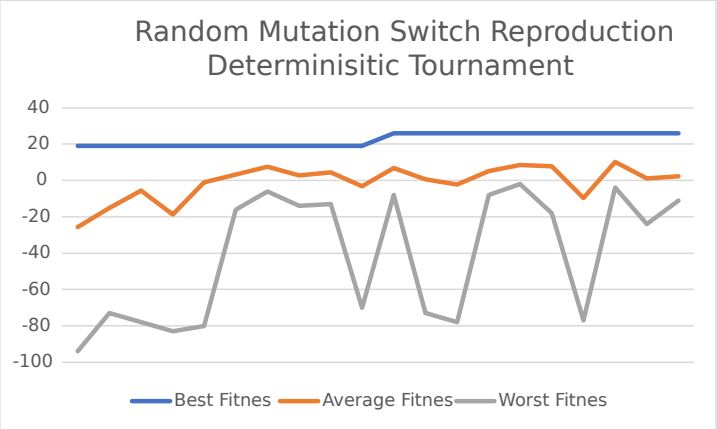

Figura 6.52: Resultados en MicroRTS | Random Mutation | SwitchCross Reproduction | Deterministic tournament.

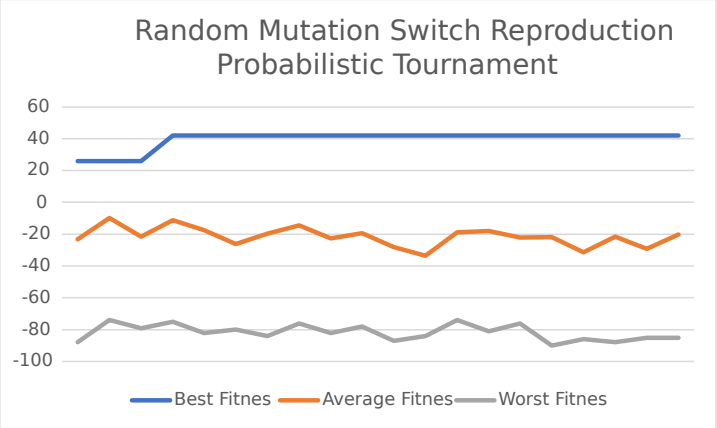

Figura 6.53: Resultados en MicroRTS | Random Mutation | SwitchCross Reproduction | Probabilistic tournament.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.53 con los parámetros:

- Método de mutación: *Random*.
- Método de reproducción: *SwitchCross*.
- Método de selección: *Probabilistic tournament*.

Los resultados generados son malos y varían a causa de la poca variedad generada por el método de reproducción *SwitchCross*, no obteniendo así una mejoría en los resultados.Tampoco varía mucho el peor a causa del método de selección. Para obtener una mejoría habría que aumentar el numero de iteraciones realizadas en la ejecución y la probabilidad de mutación.

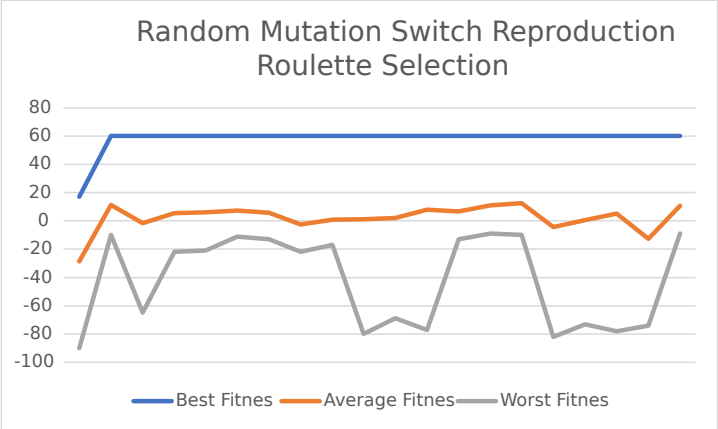

Figura 6.54: Resultados en MicroRTS | Random Mutation | SwitchCross Reproduction | Roulette.

Resultados obtenidos de ejecutar nuestra aplicación en la gráfica 6.54 con los parámetros:

- Método de mutación: *Random*.
- Método de reproducción: *SwitchCross*.
- Método de selección: *Roulette*.

Los resultados generados son malos a causa de la poca variedad generada por el método de reproducción *SwitchCross*, no obteniendo así una mejoría en los resultados. Para obtener una mejoría habría que aumentar el numero de iteraciones realizadas en la ejecución y la probabilidad de mutación.

6.2. Discusión

De los resultados obtenidos se pueden discutir múltiples cuestiones. Para empezar se ha probado la eficacia de nuestra herramienta en dos juegos de géneros completamente diferentes, ambos bastante influenciados por el azar. A pesar de poseer tanta aleatoriedad propia en el juego la herramienta logra obtener resultados de media mejores a lo normal. indicando así el correcto funcionamiento y utilidad de nuestra herramienta. A pesar de obtener buenos resultados se han encontrado muchos puntos a mejorar. Estos puntos se localizan en mejorar o rehacer las diferentes funciones y métodos utilizados en el algoritmo genético.

Gracias a las pruebas, se han obtenido resultados que se acercan bastante a los que se buscaba, logrando mejoras a las funciones implementadas en los diferentes métodos. La mayoría de los malos resultados obtenidos han sido

a causa de la poca variabilidad de ciertos métodos, los cuales sirven como muestra de cómo los métodos elegidos pueden acelerar o ralentizar en gran medida todo el proceso de evolución de una IA.

Interpretación de los datos sobre cada uno de los métodos aplicados:

6.2.1. Métodos de mutación

- *Horizontal mutation*: Los resultados obtenidos por este método no son malos, pero si mejorables. Debido a la ligera tendencia que tiene este método en generar valores extremos al aumentar los valores por el doble o dividirlos a la mitad, no es capaz de alcanzar ciertos valores, sobretodo impares y localizados cerca de la mitad superior. Aun así no fuerza en extremo permitiendo aun alcanzar múltiples valores. Para mejorar este método se puede:

  - Elegir de forma aleatoria entre multiplicar o dividir entre 2, o por valores menores a ese como puede ser 1.5 o 1.25.
  - Cambiarlo por sumar valores dentro de un rango.
  - combinarlo con un método de reproducción que esté algo más focalizado en obtener valores intermedios.

- *Proportional mutation*: Los resultados obtenidos por este método no son adecuados, aunque de vez en cuando consiguiese buenos resultados se debió principalmente a los otros dos métodos que lo acompañaban. Este método tiene una tendencia mucho más fuerte que el anterior en alcanzar los extremos, debido a que aumenta o disminuye en 4 a 8 veces su valor. Casi directamente forzando a alcanzar únicamente estos. Por culpa de este método se pueden perder muchas combinaciones con valores intermedios. Para mejorar este método se puede:

  - Elegir el numero por el que son multiplicados entre valores mucho mas pequeños, de una forma similar al horizontal.
  - Cambiarlo por elegir el valor intermedio entre los valores restantes de la mitad superior o inferior. Similar a la estrategia *Divide and Conquer*.

- *Random mutation*: Los resultados obtenidos por este método son adecuados y esperados. Este método genera un valor al azar entre los valores posibles, por lo que no provoca tendencia en los valores. No necesita mejoras, como mucho cambiar la probabilidad de que cambien un gen u otro, aunque es innecesario.

6.2.2. Métodos de reproducción

- *ColorCross reproduction*: Los resultados obtenidos por este método son adecuados y esperables, aunque es cierto que no genera suficiente variedad entre los valores enteros al no aplicarse la misma función que en los valores con decimales. La mejor manera para mejorar este método sería la aplicación de una variante mas similar a la función usada con los valores con decimales. Y mejorando la propia función usada en los valores con decimales a una que no provoque que los individuos se centralicen tanto en los valores intermedios.

- *EvenCross reproduction*: Los resultados obtenidos por este método son nefastos. Ya que no provoca una mejoría en el algoritmo, solo fuerza los valores a ir a los extremos de una manera muy significativa.

  Todos los resultados obtenidos al observar los valores dados en todos los ejemplos, son iguales. Exactamente los mismos, en el *Warrior Defense*, los valores más pequeños posibles y en el *MicroRTS* los valores más altos posibles.

  Para mejorar este método se puede:

  - No multiplicar el valor medio por el doble, si el numero es el mismo permitir variaciones ligeras y no coger el valor intermedio siempre sino valores cercanos.
  - Cambiarlo por un método completamente nuevo que mezcle los genes de los padres en medidas diferentes dependiendo de su valor.

- *SwitchCross reproduction*: Los resultados obtenidos por este método no son adecuados y tampoco esperados. La idea en la que se basa este método es el intercambio de genes.El problema fundamental encontrado ,y por lo tanto, la fuente de origen de los problemas, es la poca influencia que tiene en la variabilidad de los valores, siendo incapaz de generar nuevos. Esto provoca una dependencia muy grande de un método de mutación que genere mucha variedad y tenga una probabilidad muy alta de que ocurra. También depende mucho de un método de selección que impida que se estanque el algoritmo.

  La forma principal para mejorarlo es darle más prioridad al método de selección y mutación, o permitirle generar valores nuevos si ambos valores son los mismos.

6.2.3. Métodos de selección

Los métodos de selección implementados son:

- *Deterministic tournament.*

- *Probabilistic tournament.*
- *Roulette.*

Todos ellos han funcionado tal y como se esperaban al ser métodos ya definidos y conocidos. Como mejoras se pueden añadir más variaciones como puede ser el *Stochastic Universal Sampling*, que se podría considerar como una versión mejorada del método *roulette* al impedir que se estanquen tan rápido el algoritmo.

Capítulo 7

Conclusiones

En este capítulo se da una conclusión al trabajo realizado, valorando si se han cumplido o no los objetivos establecidos y proporcionando una línea de trabajo para el futuro desarrollo, mejora y ampliaciones de la herramienta creada.

Los objetivos del trabajo expuestos en el Capítulo 3, son los siguientes:

- Generar una IA capaz de enfrentarse a una IA básica previamente creada.
- Tiene que ser lo más genérica posible, es decir, debe de poder usarse en cualquier videojuego siempre que este se encuentre bien estructurado.
- Debe de ser fácil de usar. Con tan solo unos conocimientos muy básicos de algoritmos evolutivos debería ser posible ejecutar la herramienta y obtener un resultado.
- Todo debe de poder ser manejable desde la interfaz, sin modificar código.
- Añadir nuevos componentes como individuos, genes o métodos de selección, reproducción y mutación debe de resultar sencillo y sin necesidad de modificar el código interno de la herramienta.
- Generar árboles de decisión simples que se puedan adaptar a cualquier videojuego.

7.1. Valoración de objetivos

A continuación, se realiza un análisis y valoración de lo logrado en cada uno de los objetivos propuestos:

- **Generar una IA capaz de enfrentarse a una IA básica previamente creada.** Este objetivo hace referencia, principalmente al buen

funcionamiento de la herramienta y su capacidad de obtener buenos resultados. Este punto puede ser considerado como el primero en haber sido cumplido, al menos en el apartado de funcionalidad, al implementar los algoritmos evolutivos en la herramienta y obtener resultados en diferentes juegos de prueba. A pesar de ser lo primero en cumplirse y la base de la herramienta, tras analizar los resultados se ha observado una posible mejora en algunos de los métodos de reproducción y mutación. Esto no significa la presencia de fallos en el funcionamiento,sino resultados muy limitados en casos específicos, por lo que necesitan cambios y mejoras mencionadas en el Capítulo 6 en el apartado de discusiones.

A pesar de mencionar que algunos métodos no hayan logrado conseguir resultados adecuados, algunos de esos métodos también se han mantenido esperando que actuasen de esa manera. Para poder crear una comparativa adecuada con lo que ocurriría si no se hicieran lo suficientemente bien y genéricos. Esta planteado mejorar o eliminar estos métodos como trabajo futuro.

- **Tiene que ser lo más genérica posible, es decir, debe de poder usarse en cualquier videojuego siempre que este se encuentre bien estructurado.** A pesar de su complejidad, este objetivo se puede dar por completado, a excepción de ciertos añadidos que no se lograron terminar por falta de tiempo. Entre ellos, se pueden considerar la carga de los resultados obtenidos automáticamente. La herramienta está completamente desvinculada de los juegos. A pesar de ello, se necesita normalmente pequeñas modificaciones a los juegos para su correcto funcionamiento.

- **Debe de ser fácil de usar. Con tan solo unos conocimientos muy básicos de algoritmos evolutivos debería ser posible ejecutar la herramienta y obtener un resultado.** Se necesitan unos conocimientos básicos o nulos para poder usar la herramienta y obtener resultados. Poseyendo conocimientos superiores en algoritmos evolutivos el propio usuario de la herramienta es capaz de utilizar con eficacia la herramienta. Incluyendo la posibilidad de una mejora y personalización de la herramienta.

- **Todo debe de poder ser manejable desde la interfaz, sin modificar código.** Gracias a la decisión de evitar la necesidad de que el desarrollador escriba código para que sea de fácil acceso desde el editor de *Unity*, se ha logrado hacer una herramienta más sencilla de usar. Pero a pesar de todo sigue teniendo una complejidad en su entendimiento y uso, para el cual se aportará un manual de usuario también, por si se alojasen dudas sobre su uso. Aunque no se ha considerado necesario, inicialmente también se planteaba realizar pruebas de usuario en un

futuro para poder mejorar más la herramienta.

- **Añadir nuevos componentes como individuos, genes o métodos de selección, reproducción y mutación debe de resultar sencillo y sin necesidad de modificar el código interno de la herramienta.**

  Se ha conseguido obtener una herramienta bastante completa que permite la obtención de los recursos necesarios para el funcionamiento del algoritmo, sin la necesidad de que el usuario tenga que escribir código. Gracias al diseño de la herramienta, es muy genérica y fácilmente modificable. Los principales elementos modificables de la herramienta son los métodos de reproducción, mutación y selección que se utilizan, la función para obtener el valor de *fitness* y los múltiples parámetros que usa el algoritmo. Además se pueden agregar nuevos tipos de gen e individuos con suma facilidad. Todos estos elementos disponen de fácil

  acceso desde el editor y sin necesidad de escribir código. En el caso de crear nuevos métodos solo tendrán que heredar de una clase específica y sobrescribir la función correspondiente a ella. La nueva clase se mostrará en la interfaz una vez esta sea actualizada. Los apartados restantes por generalizar serán comentados en la sección de trabajo futuro.

- **Generar árboles de decisión simples que se puedan adaptar a cualquier videojuego.** Lamentablemente este objetivo ha sido imposible de lograr con el tiempo asignado para el proyecto, aunque ya está planteado como se cumpliría y la herramienta está preparada para que sea más sencillo implementarlo.

7.2. Trabajo futuro

Como se ha recalcado anteriormente, se han cumplido la mayoría de los objetivos dejando ciertos puntos sin completar o mejorar a causa del tiempo y la complejidad de alguno de ellos. Dentro del posible trabajo futuro que se haría en la herramienta se encuentran:

- **Mejoras y ampliaciones en los métodos de mutación, reproducción y selección**. Se ampliarían y se modificarían los métodos según se han comentado en el capítulo 6 en el apartado de discusiones. Se realizarían pruebas, para comprobar sus resultados y posibles comportamientos inesperados mejorando efectivamente la eficacia de la herramienta.

- **Acceso a los datos guardados en anteriores ejecuciones.** Permitir a través de la interfaz, el acceso y carga de los genes del mejor

individuo para evitar que el usuario lo haga manualmente. Esta mejora es necesaria para facilitar la carga de estos datos, eliminando la necesidad de insertar manualmente las variables en IA sobre la que se quieran aplicar.

- **Realizar pruebas con usuarios sobre la facilidad de uso de la herramienta.** Este periodo de pruebas serviría para mejorar la herramienta en términos de uso, sobretodo con nuevos usuarios. Estas pruebas se realizarían a través de pruebas vigiladas u observadas con voluntarios y programadores de *Unity*. Más tarde sería subido a la *Asset Store* de *Unity* de donde se obtendría una nueva retroalimentación.

- **Ampliación de la herramienta para generar Árboles de Decisión simples.** Esta ampliación estaba planteada como un objetivo. La idea es otorgarle a la herramienta la posibilidad de crear directamente una IA simple que cumpla los requisitos que pida el desarrollador, en lugar de trabajar sobre una existente. Aunque es posible crear un árbol de decisión con algoritmos evolutivos, el nivel de complejidad, tanto de la herramienta como de su desarrollo, es demasiado elevado y se necesitaría un periodo largo de tiempo para conseguirlo. Por ello se planteó como una posible expansión de la herramienta.

- **Añadir la opción de detectar el escenario y las acciones que puede realizar el individuo.** Durante el desarrollo se planteó en la generalización de la herramienta el crear *prefabs* que permitieran detectar el entorno para que resultara mucho más fácil el ofrecer la información del estado del juego a la IA. Del mismo modo, también se pensó en agregar un modo de indicar al individuo las distintas acciones que la IA puede realizar. Sin embargo, al igual que con los árboles de decisión, esta idea era demasiado ambiciosa por lo que se decidió dejar a un lado en pos de perfeccionar las otras partes de la herramienta.

Apéndice A

Contribuciones individuales

En este apartado se detalla las contribuciones de cada uno de los integrantes que han realizado este TFG.

A.1. Alejandro Ansón Alcolea

Durante el desarrollo de este trabajo he realizado diversas tareas, sobre todo de programación, pruebas y análisis de resultados. Siempre asegurando el correcto funcionamiento de la herramienta.

Como al principio no teníamos perfectamente clara la idea del TFG, mi primera tarea fue buscar información de proyectos de inteligencia artificial realizados en *Minecraft*. Debido a que fue una de las idea iniciales, en la que se planteó hacer una IA que se comportase de una manera similar a un jugador y cuya función fuera ayudarle y apoyarle. Los resultados de la investigación no fueron positivos, era un proyecto demasiado amplio y complicado.

Como continuación de esa tarea me dediqué a estudiar algoritmos genéticos y evolutivos, cuanto se suelen usar y para qué, y en cuantos juegos y herramientas de *Unity* se usan. De esta investigación se sacó el tema final y definitivo del TFG.

Tras obtener la idea, me dediqué a hacer una demo sencilla que tuviese ya el comportamiento de un algoritmo genético básico aplicado en cubos de colores.

Cuando ya tuvimos el juego *Warrior Defense* elegido, adapté el algoritmo genético para que se pudiese aplicar en el juego. Utilizando de base la demo funcional que usaba colores como genes. Por ello tuve que ampliarla añadiendo nuevos tipos de genes y el como son afectados en los diferentes métodos.

Tras terminar de adaptar el algoritmo arreglé los múltiples errores que se

encontraban en el código del que partimos en *Warrior Defense*, junto a mi compañero Maikel. Ya que a causa de ellos no se podía convertir ese proyecto en juego de verdad.

Mientras arreglábamos los errores, implementé la *IA* del *Warrior Defense* en la que se aplicaría el algoritmo genético. Siendo esta una IA muy sencilla pero funcional y además adecuada para nuestra herramienta.

Con el juego casi funcional y terminado me encargué de la creación de la demo y estudio de los resultados obtenidos con cada una de las combinaciones posibles. Los resultados obtenidos los incluí en la presentación realizada a nuestro director del TFG para el segundo hito.

Cuando Fede, nuestro tutor del TFG, nos dio el *MicroRTS* me encargué de comprobar su viabilidad para poder incluirlo como segundo juego de ejemplo.

Implementé la IA del *MicroRTS* y ayudar a corregir y arreglar todos los errores, junto a mi compañero Adrián, para asegurar el funcionamiento del juego con nuestra herramienta. Este proceso nos ocupó más tiempo del que esperábamos debido a la cantidad de cosas a cambiar y errores inesperados.

Tras cada cambio nuevo realizado por mi compañero Adrián en la herramienta al hacerla genérica, revisaba su correcto funcionamiento en ambos juegos si estos estaban funcionando correctamente en ese momento.

Como ultima aportación a la herramienta me encargué de realizar de nuevo las pruebas con los cambios aplicados, para obtener los resultados de ambos juegos,

También hice el análisis de los resultados y su documentación en la memoria en el apartado Capítulo 6 de la memoria.

De manera paralela a la realización de las tareas anteriores, también he aportado contenido a la memoria, desarrollando varios apartados de esta y ayudando a desarrollar los demás. También me he encargado de revisar la memoria en busca de errores o incongruencias, al igual que mis compañeros.

A.2. Adrián Ogáyar Sánchez

A lo largo del TFG he realizado diversas tareas de documentación, redacción y programación, así como diseño.

Comencé buscando ideas para nuestra herramienta. En concreto me dediqué a investigar *Unity ML-Agents*, una herramienta de *Unity* que permite generar IA mediante técnicas de Aprendizaje Automático. Para ello tuve que descargarlo y probar su funcionamiento hasta sacar unas conclusiones.

Tras esto, pasé a estudiar la herramienta CBR (*Case-Based Reasoning*) creada por un estudiante de doctorado de nuestra facultad, Maximiliano Miranda. Esta herramienta consistía en un algoritmo CBR en *Java* que permitía a una IA jugar a *Pac-Man* a partir de unos ejemplos previamente dados. Mi trabajo consistió en analizar esta herramienta para determinar cómo de viable era de pasar a C# para incluirlo en una herramienta de *Unity*. Finalmente se concluyó que era posible de realizar, pero que probablemente sería un trabajo demasiado simple para un TFG.

Una vez descartada la herramienta CBR, me puse a investigar sobre algoritmos evolutivos, recavando información al respecto a partir de artículos, buscando sus usos más significativos en la actualidad y comenzando lo que más tarde sería el Estado de la cuestión (2). Dado que ya tenía conocimientos de computación evolutiva, no me fue muy necesario estudiar al respecto y tan solo me dediqué a buscar artículos y ampliar mis conocimientos.

Tras buscar información general sobre computación evolutiva, pasé a buscar el uso de algoritmos evolutivos en videojuegos. También comencé a escribir los Capítulo 1 y Capítulo 2 en inglés, dado que inicialmente el TFG estaba planteado hacerse en inglés. Mientras escribía la memoria, ayudé ligeramente con el código del algoritmo evolutivo y con algunos errores menores del juego *Warrior Defense*.

Debido a que decidimos pasar a hacer el TFG en español, tuve que traducir los dos capítulos que ya estaban casi terminados de vuelta a español, si bien quedaron partes por traducir. Además seguí investigando juegos que se pudieran usar para probar la herramienta, al final descubrí dos juegos prometedores: un clon de *Counter Strike* y otro de *Darkest Dungeon*, ambos hechos en *Unity*. Sin embargo ninguno de estos juegos llegó a usarse para las pruebas.

Una vez mi compañero Alejandro finalizó el algoritmo evolutivo, pasé a generalizarlo para que fuera útil en la herramienta. Para ello diseñé la arquitectura de la herramienta, abstraje las clases del algoritmo evolutivo y creé los distintos métodos de comunicación entre la herramienta y *Unity*. Este fue un proceso largo, dado que requería abstraer muchas cosas, no solo usando *Unity*, sino muchas cosas relacionadas con C# como el *Assembly* de

las cuales no tenía a penas conocimiento. Por ello tuve que investigar muchas cosas del lenguaje como el buscar todas las clases que heredan de otra en tiempo de ejecución o ejecutar una función matemática a partir de texto introducido por el usuario.

También fue un reto hacer que el código fuera lo más abstracto posible, ya que tuve que diseñar todo basado en interfaces y clases abstractas, haciendo que todo se comunicara de la forma más sencilla posible, lo que supuso un problema en ocasiones dado que las clases plantilla no permitían hacer determinadas cosas y me obligaron a investigar más al respecto y buscar otros enfoques.

Tras terminar la generalización de la herramienta, me puse a implementarla en el nuevo juego que habíamos seleccionado, *MicroRTS*, junto a mi compañero Alejandro. Debido a la complejidad del juego y su arquitectura, esta tarea llevó algo más de tiempo del planeado, pero finalmente se logró terminar su implementación.

Cuando la herramienta estuvo lista para obtener resultados, pasé escribir los aparatados Sección 5.1 y Sección 5.2, creando también los distintos diagramas de clase de la herramienta.

Por último, una vez la memoria estuvo terminada, me ocupé de revisar los apartados de estado, objetivos, requisitos, análisis y diseño. Con los comentarios de Fede y Maxi, modifiqué gran parte de los objetivos, moví un trozo del análisis a los requisitos y lo completé, y tuve que reescribir casi todo el análisis, así como pasar casi toda la implementación escrita por mi compañero Maikel a la sección de diseño.

A.3. Maikel Jesús Spranger Hierro

Debido a que soy el único miembro del equipo que no cursó la asignatura de programación evolutiva, este proyecto supuso un reto para mi al no conocer el funcionamiento que se quería conseguir en un inicio. Por otro lado gracias a la realización de este proyecto he llegado a comprender mejor la materia y ahora dispongo de unos conocimientos de base sobre estos algoritmos.

El trabajo que realicé fue desde investigación a implementaciones de diferentes elementos de la herramienta, además de trabajar en la escritura de la memoria y su edición como vamos a describir a continuación.

Al igual que mi compañero Adrián empecé por investigar y hacer pruebas sobre la herramienta de *Unity ML-Agents*, tal como sugirió nuestro director de proyecto, ya que tenía un funcionamiento similar al que se quería conseguir con la herramienta que queríamos desarrollar. Tras investigar esta herramienta se llego a la conclusión de que si bien tenía un funcionamiento similar a lo que queríamos conseguir, no implementaba algoritmos evolutivos por lo que podríamos conseguir unos resultados diferentes con nuestra herramienta.

A continuación me dediqué a la búsqueda de videojuegos *open source* sobre los cuales poder hacer las pruebas con nuestra herramienta. En un inicio hice una lista con los videojuegos más prometedores que había encontrado y pertenecían a diferentes géneros, *Roman Defenders, Warrior Defense, juegos de lucha.* los cuales se discutieron en una de las reuniones para decidir cuál seleccionar.

El juego seleccionado fue *Warrior Defense* por ser el que mejor estaba diseñado, en cuanto a su uso e implementación con el patrón de diseño factoría. Una vez seleccionado este juego, empecé a hacer pruebas sobre él en mayor profundidad para comprobar su funcionamiento. Por desgracia al realizar estas pruebas encontré que el funcionamiento era muy limitado y necesitaría una buena cantidad de cambios para poder ser considerado un videojuego y ser útil para las pruebas de nuestra herramienta.

Ya que la selección de videojuegos *open source* que se pueden encontrar es amplía, pero no de gran calidad debido a su uso comercial, al no disponer de demasiado tiempo para empezar a realizar la herramienta decidimos continuar con este videojuego y añadir todas las funcionalidades necesarias. Junto a mi compañero Alejandro implementé estas funcionalidades. Yo me encargué de que el videojuego tuviera un sistema de puntuaciones y una jugabilidad básica, mientras que mi compañero se centraba en su IA. También añadí una interfaz de usuario básica para el videojuego además de resolver diferentes fallos que fui encontrando.

Durante la revisión de estas funcionalidades encontré algunos elementos que podrían funcionar mejor, como la barra de mana o energía o el equilibrio base entre los personajes luchadores que se invocan para que no simplemente se enfrenten todos en igualdad de condiciones sino que algunos sean mejores que otros y viceversa pero que esto se vea reflejado en su coste de invocación, su velocidad y su daño.

Al investigar videojuegos del mismo tipo que *Warrior Defense*, descubrí que algo común en estos es la gran cantidad de personajes que se pueden invocar, esto le da más profundidad al juego en cuanto a combinaciones que se pueden realizar. Por esto añadí luchadores nuevos dando así un total de doce personajes luchadores al contrario de los seis iniciales, permitiéndonos conseguir muchas más posibilidades a la hora de invocar y resultados más variados a la hora de ejecutar el algoritmo evolutivo.

Una vez mi compañero Alejandro terminó de implementar la base del algoritmo evolutivo, me dediqué a estudiar su funcionalidad e implementación para poder expandirlo en cuanto a métodos de selección, reproducción y mutación, implementando así diferentes variantes para estos métodos que nos permitirán conseguir más combinaciones de resultados.

Ya teniendo el algoritmo evolutivo funcionando para el videojuego *Warrior Defense* y generando buenos resultados, me dedique a documentar la implementación de tanto el algoritmo, la herramienta y los cambios sobre este primer videojuego seleccionado para las pruebas en la memoria.

Sobre el trabajo realizado en la memoria, además de los apartados de diseño e implementación de los componentes de la herramienta, y el manual de usuario también me encargué de, realizar una traducción completa de la introducción al inglés debido a que se hicieron diferentes cambios en la versión en español desde que se agregó en un inicio, expandir la sección de metodología que era muy limitada en un inicio, realizar el resumen de nuestro proyecto, repasar y terminar la versión en inglés de la sección de conclusiones, además de revisar la memoria en general para corregir fallos gramaticales y expandir en puntos concretos.

Appendix B

Introduction

This chapter introduces the subject of the TFG, the scope and the purpose of it. In addition, the structure of this document is presented.

Balance between different components of a videogame has been debated in past few decades since they first showed up. A weapon being too powerful, an enemy having too much health, the reach of a character moves... All these parameters could make the game either too easy or too difficult for the players, making an unpleasurable experience for them.

This makes the developers have to invest hundreds of hours just into balancing their games, either by having a team working straight into this, hiring testers or setting up alpha/beta testing private or public for players to make sure the game works as intended and also help them find the balance between the components. In most solo player games, a weapon that's too powerful or too weak, an enemy that has too much health might not be a problem if the player has the option to pick something else or the enemy, although being more difficult, can be defeated in a longer but small period of time and won't lead to the player's frustration. However, in player versus player games (PvP) having balance it's essential, having a stronger weapon or character would make the players feel pressured to either pick them up or have a disadvantage agaisnt other players, limiting the options of the players, thus creating a meta which most players would follow, making the game repetitive and with only a few viable strategies.

Therefore, testing the balance of a videogame is a process that involves a huge investment of resources, but it's also necesary since avoiding this step of the development process would lead to failure in most cases.

As the game complexity increases, so does the difficulty of testing all the possibilities it offers, requiring more time investment, and sometimes, making the developers launch betas to get the most number of players and make sure most of the combinations the game offers are tested.

This issue increases exponentialy when there's need to balance Artificial Inteligence (AI) in a game. When the AI is poorly implemented might lead

to direct player frustration or the lost of all of the game's difficulty. It's also common practice to make multiple levels of difficulty, and depending on the game it's necesary to modify this AI, causing unexpected changes in the user's experience of the game. Unlike having a stronger weapon that could still be used by the players, a small change on the AI parameters could lead to not being able to clear the game at all.

While this may not be a problem for large studios, it is a problem for smaller studios with smaller budgets. Investing hundreds or thousands of euros in videogame testing will be impossible for many developers and, although the developers themselves can work on this, it means spending hundreds of hours playing and testing different strategies.

With the rise of free videogame engines (or royalty systems) and the indie industry, the number of low budget games have increased greatly, and thus creating the need for new approaches for videogame testing without having to rely on human testers, reducing the cost and development times.

For this reason, the goal of this project is to research the development of a tool designed for the Unity game engine, using Evolutionary Algorithms that are capable of testing the game and automate the obtaining of the parameters for the AI according to the designer's choice.

B.1. Scope

Even thought it's posible to design a tool for Unity in particular for a type of videogame, our final goal is to create a piece of software that works with any kind of videogame regardless of it's genre or features, that the developers can use without having great knowledge of programming or algorithms. However, this implies generating an AI capable of learning to play any kind of videogame even it's just the basic mechanics, which is a reachable goal in the long term, but involves a lot of work that is not contemplated in this project, more focused in testing and use of the Evolutionary Algorithms.

Thus, the tool will be designed using an open source game, Warriors Defense, to analyze it's performance. Will also try to introduce the tool into games of different genres to test it's efficiency. Making necesary to create an AI capable of playing the selected game, and an Evolutionary Algorithm that allows to randomize the parameters and keep the best posible combinations to obtain the desired result by the designer.

B.2. Purpose

The main issue we're trying to solve is the huge load of time and resources, both human and computational, that causes the balancing of AI in videogames. This issue is accentuated in inexperienced developers and our

goal is to help them speed up the process and improve their results, all with as little input and as much automation as possible.

B.3. Work structure

This work is structured as follows. In this Appendix B the scope and purpose of this Final Degree Project is presented. In the chapter 2 the state of the art in Artificial Intelligence used in videogames or as a technique to help in the development of videogames is reviewed. In the chapter 3 the objectives are reviewed in detail, considering the corresponding specifications of the tool, taking into account restrictions, functions and user characteristics. In the chapter 4 the tools used are presented as well as the planning that has been followed in the development of the TFG. In the chapter 5 an analysis of the objectives and specifications is made, the design of the EvoUnity tool is also presented, as well as the implementation and testing issues encountered during the development. In the chapter 6 all the results of the tests carried out in both games are presented, as well as the discussion about them. Finally, it ends with the Appendix C where the conclusions of the project are summarized, and what has been achieved with respect to the objectives.

Appendix C

Conclusions

This chapter provides a conclusion to the work achieved, considering whether or not the established objectives have been met and providing a work plan for the future development, improvements and extensions for the resulting tool.

The objectives of the work previously stated in chapter 3 are as follows:

- Generate an AI capable of facing a basic AI previously created.
- It has to be as generic as possible, in other words, it must be able to be used in any videogame as long as it is well structured.
- It must be easy to use. With only a very basic knowledge of evolutionary algorithms it should be possible to execute the tool and obtain results.
- Everything must be handled from the interface, without modifying code.
- Adding new components such as individuals, genes or selection, reproduction and mutation methods should be easy and without the need to modify the tool's internal code.
- Generate simple decision trees that can be adapted to any video game.

C.1. Objective analysis

Next, an analysis and evaluation of what has been achieved in each of the previously proposed objectives, will be carried out:

- **Generate an AI capable of facing a basic AI previously created.**. This objective referred mainly to the performance of the tool and its ability to obtain good results. This point can be considered as

the first to have been fulfilled, at least in the functionality section, by implementing the evolutionary algorithms in the tool and obtaining results in different test sets. Although it is the first to be fulfilled and the basis of the tool, after the observation and analysis of the results, a possible improvement in some of the reproduction and mutation methods has been realized. This does not mean the existence of malfunctions, but in specific cases they have provided very limited results, so they would need the changes and improvements already mentioned in the chapter 6 in the discussions section.

Despite mentioning that some methods have not achieved proper results, they have been kept expecting to perform that way in order to show contrast with better defined and more generic methods. It is planned to improve or remove these methods as future work.

- **It has to be as generic as possible, in other words, it must be able to be used in any videogame as long as it is well structured.** This point has been complicated to accomplish, but it may be considered as done, with the exception of certain additions that we have not been able to do because of the lack of time. Among them, it can be considered the automatic loading of the results obtained. The tool is completely independent of the games. However, small modifications to the games are usually necessary for its correct performance.
- **It must be easy to use. With only a very basic knowledge of evolutionary algorithms it should be possible to execute the tool and obtain results.** Basic or non-existing knowledge is required to be able to use the tool and obtain results. By having a superior knowledge in evolutionary algorithms, the user of the tool is also able to use the tool effectively. Including the possibility to improve and customize the tool.
- **Everything must be handled from the interface, without modifying code.** The tool was made more accessible and simpler to use throught *Unity's* editor due to the decision to avoid developers from writing code. But despite this it still has some complexity in its use and understanding, therefore a user manual will be provided as well, in case there are any doubts about how to use it. Although initially it was not considered necessary, in the future it's contemplated to carry out user testing in order to further improve the tool.
- **Adding new components such as individuals, genes or selection, reproduction and mutation methods should be easy and without the need to modify the tool's internal code.**

  The tool allows users to set the resources needed to execute the evolutionary algorithm, without having to write any kind of code. In spite

of making it generic, it is still easily customizable. The main customizable elements of the tool are the reproduction, mutation and selection methods used, the function to obtain the *fitness* value and the multiple parameters used by the algorithm.

All these elements are easily accessible from the editor without the need to write code. In case of creating new methods these will only have to inherit from a specific class and overwrite the function corresponding to it. The new class will be added to the interface after updating it. The remaining sections to be generalized will be discussed in the future work section.

- **Generate simple decision trees that can be adapted to any video game.** Unfortunately this objective has been impossible to achieve with the time available for the project, although it is already planned how it would be accomplished and the tool is prepared to make it easier to achieve it.

C.2. Future work

As previously highlighted, most of the objectives have been met or almost met, leaving certain points incomplete or to be improved due to a lack of time. Within the possible future work to be done on the tool are:

- **Improvements and extensions in mutation, reproduction and selection methods**. The methods would be extended and modified as discussed in the chapter 6 in the discussion section. Tests would be performed, to check their results and possible unexpected behaviors, effectively improving the efficiency of the tool.
- **Access to the data saved in previous executions.** To avoid users from copying data straight from the result files to their AI parameters, the user interface would allow to load this data straight away.
- **Usability testing of the tool.** This testing period would serve to improve the tool in terms of usability, especially with new users. This testing would be done through monitored or supervised testing with volunteers and programmers from *Unity*. Later the tool would be uploaded to the *Unity's asset store* to obtain further feedback.
- **Extend the tool to generate simple Decision Trees.** This extension was proposed as an objective. The idea of this extension is to give the tool the possibility to directly create a simple AI that meets the developer's requirements, instead of working on an existing one. Although it is possible to implement this functionality with evolutionary

algorithms, the level of complexity becomes much higher and it would take a long period of time to achieve it.

- **Add an option to detect the scene and actions allowed to the individual.** During the development of the tool the option to create prefabs that allow to detect the enviroment of the videogame was raised, this would make much easier to translate the state of the game to the evolutionary AI. Likewise for the option to indicate to the individual of the actions that the videogame AI could perform. However, as with the previous objective, this idea was too ambitious so it had to be posponed in order to perfect the rest of the tool.

Apéndice D

Manual de uso

En este apartado se va a explicar el uso la herramienta en *Unity* utilizando como ejemplo el videojuego *Warrior Defense*.

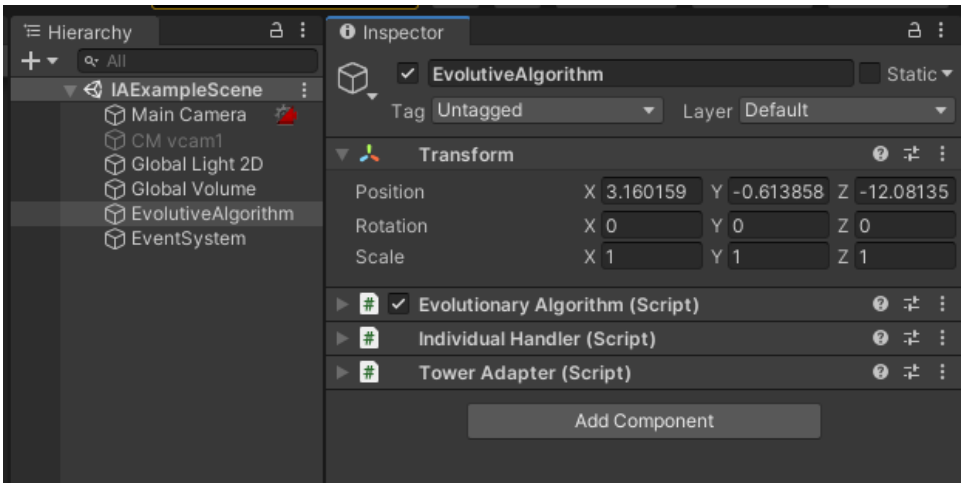

Figura D.1: Objeto EvolutionaryAlgorithm el cuál creará la instancia del juego y ejecutará el algoritmo.

El usuario tendrá que crear un objeto vacío en una escena de *Unity* (EvolutionaryAlgorithm) y luego se le añadirán los siguientes scripts:

- Evolutionary Algorithm (Figura D.2): Este *script* permite al usuario seleccionar todas las variables relacionadas con el funcionamiento del algoritmo evolutivo, cada variable llevará a un resultado distinto y el usuario tendrá que realizar pruebas para encontrar los resultados más convenientes para su caso en específico.

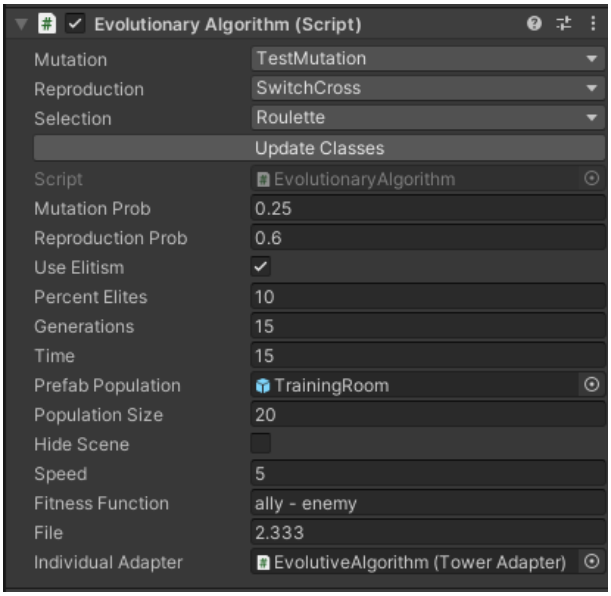

Figura D.2: Script EvolutionaryAlgorithm.

- *Mutation*: Permite seleccionar el método de mutación que ejecutará el algoritmo evolutivo.
- *Reproduction*: Permite seleccionar el método de reproducción que ejecutará el algoritmo evolutivo.
- *Selection*: Permite seleccionar el método de selección que ejecutará el algoritmo evolutivo.
- *Update Classes*: Botón que permite actualizar la lista de los métodos de mutación, reproducción y selección en caso de que el usuario añada o elimine alguno de estos.
- *Mutation Prob*: Valor de probabilidad para que ocurra la mutación de un gen.
- *Reproduction Prob*: Valor de probabilidad para que ocurra la reproducción de dos genes.
- *Use Elitism*: Permite elegir si se desea el uso o no de elitismo, lo que permite que se guarden los mejores individuos de generaciones pasadas y no se pierdan debido a la aleatoriedad, puede ser útil en algunos casos pero esto hace que se reduzca la diversidad de los resultados.
- *Percent Elites*: Permite elegir el porcentaje de individuos elite que se seleccionarán, en caso de ser positiva la opción anterior.
- *Generations*: Número de generaciones o iteraciones que realizará el algoritmo evolutivo.

- *Time*: Tiempo de duración de una generación, en el cuál se generarán los nuevos resultados.
- *Prefab Population*: *Prefab* creado previamente por el usuario que contiene la escena de su videojuego. En el caso de *Warrior Defense* el *prefab* creado ha sido *TrainingRoom*.

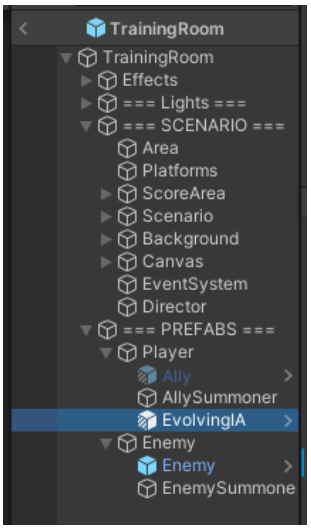

Figura D.3: Training Room Prefab.

Para que este *prefab* (Figura D.3) pueda ser utilizado con el algoritmo evolutivo debe contener dos IA's, una básica que tenga el comportamiento deseado pero sin un buen equilibrado, además de una IA a la que enfrentará y que será asignada como IA evolutiva (Figura D.6). Esta IA evolutiva funciona exactamente igual a la IA anterior pero partirá de unos parámetros aleatorios y cambiará en base a los resultados que devuelve el algoritmo a través del *script Tower Adapter*.

- *Population Size*: Número de individuos que se instanciarán durante la ejecución del algoritmo. Mientras mayor sea el tamaño de la población, mayor será el número de resultados, por lo que se podrán encontrar mejores resultados en menos iteraciones o generaciones.
- *Hide Scene*: Permite ocultar los objetos en escena durante la ejecución para reducir el consumo de recursos.
- *Speed*: Permite ajustar la velocidad de ejecución del juego, ayuda a conseguir una mayor cantidad de resultados en menor tiempo.
- *Fitness Function*: Permite introducir la función de *fitness* mediante texto, en este caso la función será una resta entre los valores *ally* y *enemy*, representando los *scores* conseguidos por estos.

- *File*: Nombre que se concatena al fichero de resultados, en este caso es un número que representa la combinación de los métodos evolutivos.
- *Individual Adapter*: *Script* creado por el usuario para adaptar los datos devueltos por el algoritmo a su objeto con IA. En el caso de *Warrior Defense* se creo el *script Tower Adapter*.

- Individual Handler (Figura D.4): En este *script* se seleccionará el tipo de datos con los que cuentan los individuos, en este caso *FloatInt*, lo que permitirá añadir los rangos de valores para los genes pertenecientes a los individuos.

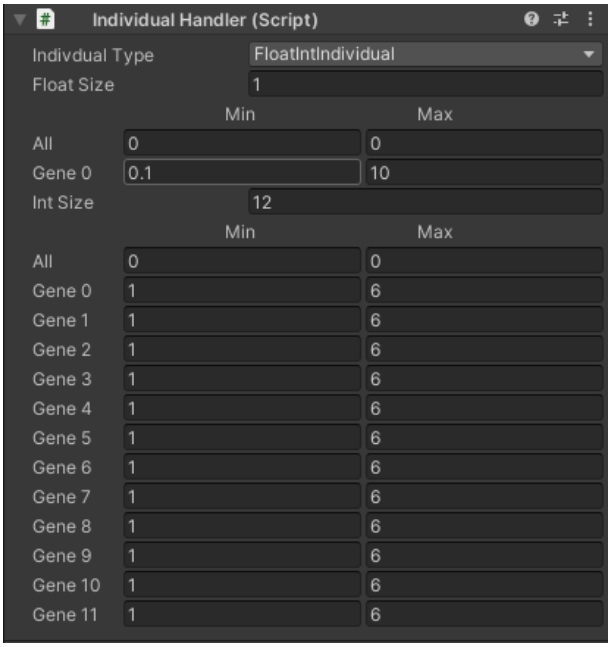

Figura D.4: Script IndividualHandler

- Tower Adapter (Figura D.5): Es un *script* específico para este videojuego (*Warrior Defense*) que permite añadir un *prefab*, el cuál será un objeto que contiene la IA que se quiere mejorar. Este *script* se encarga de adaptar los valores de los individuos de la generación a valores que puede utilizar la IA (Ej: Figura D.6).

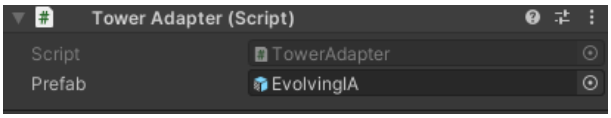

Figura D.5: Script TowerAdapter

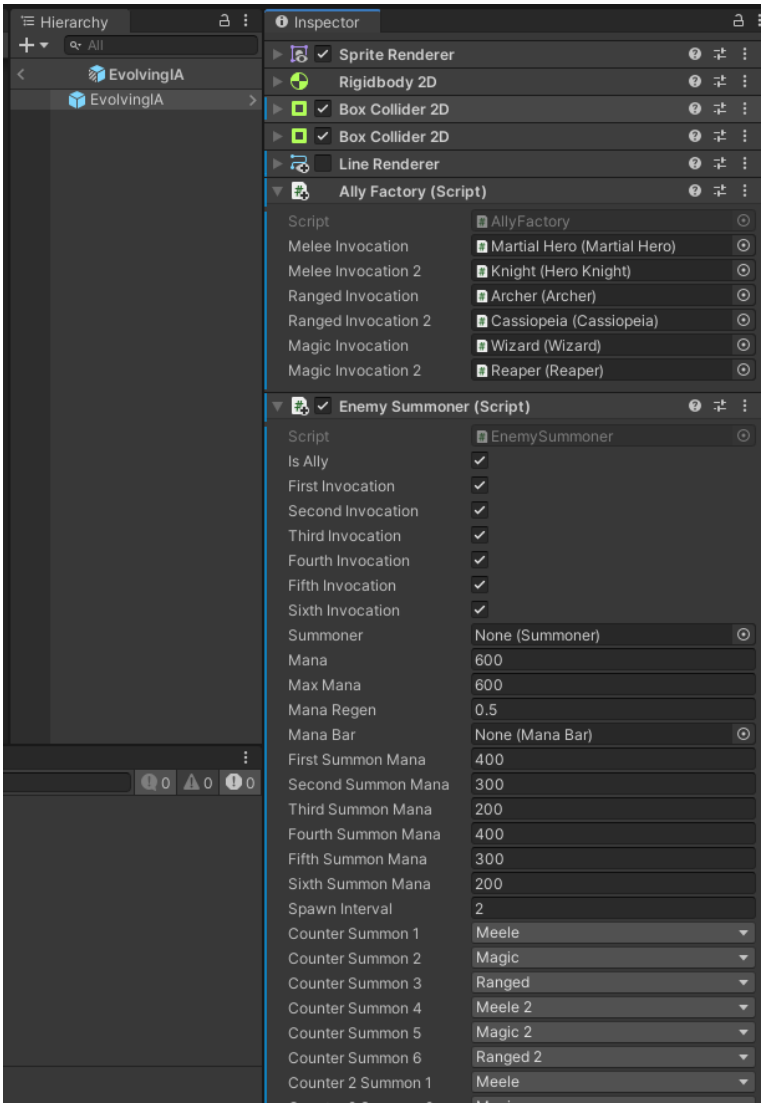

Figura D.6: Prefab Evolving IA

Por último hay que añadir los *Score Observer*, que permitirán indicar al algoritmo de dónde obtener los valores para las variables del *fitness*. Para ello, hay que colocar un *ScriptScore* en un objeto que contenga un *script* con una función/variable pública que devuelva el valor en cuestión, o un *TextAreaScore* en un *TextArea* que muestre el valor (como puede ser un contador de puntuación, por ejemplo), también se puede crear una nueva clase que herede de *ScoreObserver* y definir el modo en el que se quiere extraer la información.

Estos *Score Observers* disponen de una variable *label* que permite identificarlos. Este identificador debe de ser el mismo que se use en la función

de *fitness*, cualquier identificador que no exista en la función de *fitness* será ignorado.

En el caso de *Warrior Defense* se utilizó *TextAreaScore* (Figura D.7), basta con incluir el *label* del score y este *script* se ocupará de obtener el valor a partir del texto.

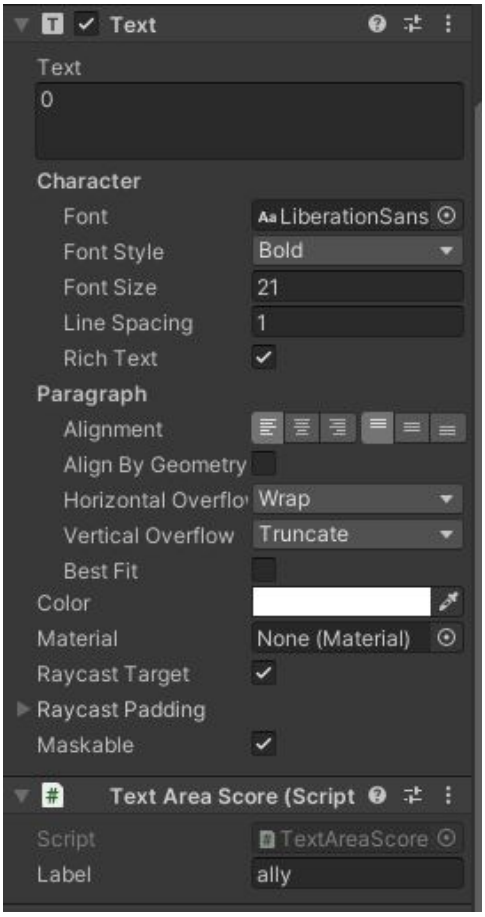

Figura D.7: Text Area Score

Por otro lado, en el segundo juego de prueba *MicroRTS* se utilizó el *ScriptScore* (Figura D.8), que tiene una serie de opciones:

- Script: Permite seleccionar el *script* que contiene el valor que se desea obtener.
- Field or Method: Indica si el valor está contenido en una variable pública o en un método.
- Getter or Counter: *Getter* indica que el valor está contenido en un método que devuelve un valor. En el caso de *Counter*, no se obtendrá

el valor a partir de dicha función, sino del número de veces que se haya ejecutado esta.

- Method (o Field): Indica el nombre del método (o variable) que hay que usar para obtener el valor.

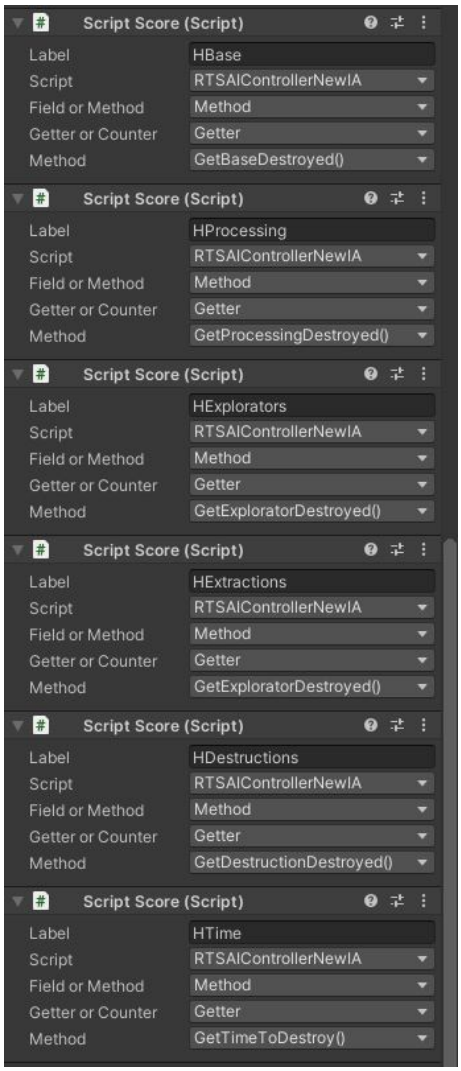

Figura D.8: Score Observer

Una vez añadidos los *scripts* correctamente, el usuario debe pulsar el botón *Play* en la escena de *Unity* y permitir al algoritmo ejecutarse durante todas las generaciones especificadas.

Al terminar la ejecución el algoritmo devolverá un fichero con los resultados, incluyendo los valores de cada gen del mejor individuo. Estos resultados ya se podrán añadir al objeto con IA del videojuego para que se disponga de

un nivel de IA similar al del resultado. Además también puede encontrar un fichero de tipo CSV, el cuál le permitirá generar diagramas donde se podrá apreciar la evolución de la IA.

Bibliografía

Aguilar, W., Santamaria-Bonfil, G., Froese, T. y Gershenson, C. The Past, Present, and Future of Artificial Life. *Frontiers in Robotics and AI*, vol. 1, 2014.

AlphaStar Team, T. AlphaStar: Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Deep Mind Blog*, 2019a.

AlphaStar Team, T. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. *Deep Mind Blog*, 2019b.

Baier, H. y Cowling, P. I. Evolutionary MCTS for Multi-Action Adversarial Games. En *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, páginas 1–8. 2018.

Beyer, H. Evolution strategies. *Scholarpedia*, vol. 2(8), página 1965, 2007. Revision #193589.

Bäck, T., Rudolph, G. y Schwefel, H.-P. Evolutionary programming and evolution strategies: Similarities and differences. *Proceedings of the Second Annual Conference on Evolutionary Programming*, 1997.

Chen, S. H., Jakeman, A. J. y Norton, J. P. Artificial Intelligence techniques: An introduction to their use for modelling environmental systems. *Mathematics and Computers in Simulation*, vol. 78(2), páginas 379 – 400, 2008. ISSN 0378-4754. Special Issue: Selected Papers of the MSSANZ/IMACS 16th Biennial Conference on Modelling and Simulation, Melbourne, Australia, 12-15 December 2005.

Drachen, A., Canossa, A. y Yannakakis, G. Player modeling using self-organization in Tomb Raider: Underworld (Pre-print). En *CIG2009 - 2009 IEEE Symposium on Computational Intelligence and Games*, páginas 1 – 8. 2009.

Fogel, G. B., Fogel, D. y Fogel, L. Evolutionary programming. *Scholarpedia*, vol. 6(4), página 1818, 2011. Revision #137293.

Frade, M., Vega, F. y Cotta, C. Breeding terrains with genetic terrain programming: The evolution of terrain generators. *Int. J. Computer Games Technology*, vol. 2009, 2009.

García-Sánchez, P., Tonda, A., Fernández-Leiva, A. J. y Cotta, C. Optimizing Hearthstone agents using an evolutionary algorithm. *Knowledge-Based Systems*, vol. 188, página 105032, 2020. ISSN 0950-7051.

García-Sánchez, P., Tonda, A., Mora, A. M., Squillero, G. y Merelo, J. J. Automated playtesting in collectible card games using evolutionary algorithms: A case study in Hearthstone. *Knowledge-Based Systems*, vol. 153, páginas 133 – 146, 2018. ISSN 0950-7051.

Kenneth O., S. Welcoming the Era of Deep Neuroevolution. *Uber*, 2017.

Langton, C. G., Taylor, C., Farmer, D. y Rasmussen, S. *Artificial Life II*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edición, 1991. ISBN 0201525704.

Lucas, S. M., Samothrakis, S. y Pérez, D. Fast Evolutionary Adaptation for Monte Carlo Tree Search. En *Applications of Evolutionary Computation* (editado por A. I. Esparcia-Alcázar y A. M. Mora), páginas 349–360. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. ISBN 978-3-662-45523-4.

McCarthy, J. What is Artificial Intelligence? *Stanford University*, 2004.

McCulloch, W. y Pitts, W. A logical calculus of the idea immanent in nervous activity. *Bulletin of Mathematical Biology*, vol. 5, páginas 115–133, 1943.

Miranda, M., Sánchez-Ruiz, A. A. y Peinado, F. A neuroevolution approach to imitating human-like play in ms. pac-man video game. Informe técnico, Complutense University of Madrid, 2016.

Pedersen, C., Togelius, J. y Yannakakis, G. Modeling Player Experience for Content Creation. *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 2, páginas 54 – 67, 2010.

Thengade, A. y Dondal, R. Genetic algorithm – survey paper. *IJCA Proc National Conference on Recent Trends in Computing, NCRTC*, vol. 5, 2012.

Turing, A. M. Computing machinery and intelligence. *Mind*, vol. 59(236), páginas 433–460, 1950.

Wexler, J. Artificial Intelligence in Games : A look at the smarts behind Lionhead Studio ' s "Black and White" and where it can and will go in

the future. *CS242 2002 – Artificial Intelligence. University of Rochester*, 2002.

Wong, H.-S. y Guan, L. Application of evolutionary programming to adaptive regularization in image restoration. *IEEE Transactions on Evolutionary Computation*, vol. 4(4), páginas 309–326, 2000.

Xu, Q.-z. y Wang, L. Recent advances in the artificial endocrine system. *Journal of Zhejiang University SCIENCE C*, vol. 12(3), páginas 171–183, 2011. ISSN 1869-196X.

Yang, X.-S. A New Metaheuristic Bat-Inspired Algorithm. *Nature Inspired Cooperative Strategies for Optimization (NICSO 2010)*, vol. 284, 2010.

Yannakakis, G. Game AI revisited. *CF '12 - Proceedings of the ACM Computing Frontiers Conference*, 2012.